SỞ LAO ĐỘNG - THƯƠNG BINH VÀ XÃ HỘI HÀ NỘI

TRƯỜNG TRUNG CẤP CÔNG NGHỆ VÀ DU LỊCH HÀ NỘI

GIÁO TRÌNH

Mô đun: Cấu trúc dữ liệu và giải thuật

NGHỀ: CÔNG NGHỆ THÔNG TIN

TRÌNH ĐỘ: TRUNG CẤP

(Ban hành kèm theo Quyết định số: /QĐ-CNDL ngày 03 tháng 06 năm 2019 của Hiệu trưởng Trường Trung cấp Công nghệ và Du lịch Hà Nội)

Hà Nội, năm 2019

TUYÊN BỐ BẢN QUYỀN

LỜI GIỚI THIỆU

Kiến thức môn học Cấu trúc dữ liệu và giải thuật là một trong những nền tản cơ bản của những người muốn tìm hiểu sâu về Công nghệ thông tin đặt biệt đối với việc lập trình để giải quyết các bài toán trên máy tính điện tử. Các cấu trúc dữ liệu và các giải thuật được xem như là 2 yếu tố quan trọng nhất trong lập trình, đúng như câu nói nổi tiếng của Niklaus Wirth: Chương trình = Cấu trúc dữ liệu + Giải thuật (Programs = Data Structures + Algorithms). Nắm vững các cấu trúc dữ liệu và các giải thuật là cơ sở để sinh viên tiếp cận với việc thiết kế và xây dựng phần mềm cũng như sử dụng các công cụ lập trình hiện đại.

Cấu trúc dữ liệu có thể được xem như là 1 phương pháp lưu trữ dữ liệu trong máy tỉnh nhằm sử dụng một cách có hiệu quả các dữ liệu này. Và để sử dụng các dữ liệu một cách hiệu quả thì cần phải có các thuật toán áp dụng trên các dữ liệu đó. Do vậy, cấu trúc dữ liệu và giải thuật là 2 yếu tố không thể tách rời và có những liên quan chặt chẽ với nhau. Việc lựa chọn một cấu trúc dữ liệu có thể sẽ ảnh hưởng lớn tới việc lựa chọn áp dụng giải thuật nào.

Về nguyên tắc, các cấu trúc dữ liệu và các giải thuật có thể được biểu diễn và cài đặt bằng bất cứ ngôn ngữ lập trình hiện đại nào. Tuy nhiên, để có được các phân tích sâu sắc hơn và mô phạm, có kết quả thực tế hơn, chúng tôi đã sử dụng ngôn ngữ tựa Pascal để minh hoạ cho các cấu trúc dữ liệu và thuật toán.

Mặc dầu có rất nhiều cố gắng, nhưng không tránh khỏi những khiếm khuyết, rất mong nhận được sự đóng góp ý kiến của độc giả để giáo trình được hoàn thiện hơn.

Hà Nội, tháng 03 năm 2019

MỤC LỤC

TRANG

GIÁO TRÌNH MÔN HỌC/MÔ ĐUN

Tên môn học/mô đun: CẤU TRÚC DỮ LIỆU VÀ GIẢI THUẬT

Mã môn học/mô đun: MH09

Vị trí, tính chất, ý nghĩa và vai trò của môn học/mô đun:

- Vị trí: sau khi học xong các môn học Tin học, Lập trình căn bản
- Tính chất: Cấu trúc dữ liệu và giải thuật là môn cơ sở nghề bắt buộc.
- Ý nghĩa và vai trò của môn học/mô đun:

Mục tiêu của môn học/mô đun:

- Kiến thức:
  - Hiểu được mối quan hệ giữa cấu trúc dữ liệu và giải thuật trong việc xây dựng chương trình;
  - Hiểu được ý nghĩa, cấu trúc, cách khai báo, các thao tác của các loại cấu trúc dữ liệu: mảng, danh sách liên kết, cây và các giải thuật cơ bản xử lý các cấu trúc dữ liệu đó;
- Kỹ năng:
  - Xây dựng được cấu trúc dữ liệu và mô tả tường minh các giải thuật cho một số bài toán ứng dụng cụ thể;
  - Cài đặt được một số giải thuật trên ngôn ngữ lập trình C;
- Về năng lực tự chủ và trách nhiệm:
  - Coi việc học môn này là một nền tảng cho các môn học chuyên môn tiếp theo, nghiêm túc và tích cực trong việc học lý thuyết và làm bài tập, chủ động tìm kiếm các nguồn tài liệu liên quan đến môn học.

BÀI 1: THIẾT KẾ VÀ PHÂN TÍCH GIẢI THUẬT

Mã bài: MH09 - 01

Giới thiệu:

Tổng quan về giải thuật. Đầu tiên là cách phân tích 1 vấn đề, từ thực tiễn cho tới chương trình, cách thiết kế một giải pháp cho vấn đề theo cách giải quyết bằng máy tính. Tiếp theo, các phương pháp phân tích, đánh giá độ phức tạp và thời gian thực hiện giải thuật cũng được xem xét trong chương.

Mục tiêu:

- Hiểu được mối quan hệ giữa cấu trúc dữ liệu và giải thuật;
- Biết được các cách tư duy về tiến trình phân tích và thiết kế thuật toán;
- Biết cách đánh giá độ phức tạp thuật toán;
- Hiểu được một số giải thuật cơ bản;
- Viết tường minh một số giải thuật;
- Nghiêm túc, tỉ mỉ trong việc học và vận dụng vào làm bài tập.

Nội dung chính:

1. Mở đầu

Có thể nói rằng không có một chương trình máy tính nào mà không có dữ liệu để xử lý. Dữ liệu có thể là dữ liệu đưa vào (input data), dữ liệu trung gian hoặc dữ liệu đưa ra (output data). Do vậy, việc tổ chức để lưu trữ dữ liệu phục vụ cho chương trình có ý nghĩa rất quan trọng trong toàn bộ hệ thống chương trình. Việc xây dựng cấu trúc dữ liệu quyết định rất lớn đến chất lượng cũng như công sức của người lập trình trong việc thiết kế, cài đặt chương trình.

2. Thiết kế giải thuật

Khái niệm giải thuật hay thuật giải mà nhiều khi còn được gọi là thuật toán dùng để chỉ phương pháp hay cách thức (method) để giải quyết vấn đề. Giải thuật có thể được minh họa bằng ngôn ngữ tự nhiên (natural language), bằng sơ đồ (flow chart) hoặc bằng mã giả (pseudo code). Trong thực tế, giải thuật thường được minh họa hay thể hiện bằng mã giả tựa trên một hay một số ngôn ngữ lập trình nào đó (thường là ngôn ngữ mà người lập trình chọn để cài đặt thuật toán), chẳng hạn như C, Pascal, ?

Khi đã xác định được cấu trúc dữ liệu thích hợp, người lập trình sẽ bắt đầu tiến hành xây dựng thuật giải tương ứng theo yêu cầu của bài toán đặt ra trên cơ sở của cấu trúc dữ liệu đã được chọn. Để giải quyết một vấn đề có thể có nhiều phương pháp, do vậy sự lựa chọn phương pháp phù hợp là một việc mà người lập trình phải cân nhắc và tính toán. Sự lựa chọn này cũng có thể góp phần đáng kể trong việc giảm bớt công việc của người lập trình trong phần cài đặt thuật toán trên một ngôn ngữ cụ thể.

3. Phân tích giải thuật

Mối quan hệ giữa cấu trúc dữ liệu và Giải thuật có thể minh họa bằng đẳng thức:

Cấu trúc dữ liệu + Giải thuật = Chương trình

Như vậy, khi đã có cấu trúc dữ liệu tốt, nắm vững giải thuật thực hiện thì việc thể hiện chương trình bằng một ngôn ngữ cụ thể chỉ là vấn đề thời gian. Khi có cấu trúc dữ liệu mà chưa tìm ra thuật giải thì không thể có chương trình và ngược lại không thể có Thuật giải khi chưa có cấu trúc dữ liệu. Một chương trình máy tính chỉ có thể được hoàn thiện khi có đầy đủ cả Cấu trúc dữ liệu để lưu trữ dữ liệu và Giải thuật xử lý dữ liệu theo yêu cầu của bài toán đặt ra.

3.1. Phân tích tính đúng đắn Thiết kế xong một thuật toán câu hỏi luôn luôn phải có đó là thuật toán được thiết kế đã đúng chưa? Cách đơn giản nhất mà được sử dụng thông dụng đó là viết chương trình cho thuật toán đã thiết kế và chạy thử chương trình với nhiều bộ dữ liệu vào cụ thể (tests) để kiểm tra dữ liệu ra có chuẩn xác hay chưa. Tuy nhiên, cách này cũng chỉ khẳng định được thuật toán đúng với các trường hợp cụ thể mà thôi. Có một cách khác chứng minh được thuật toán đúng đó là chứng minh bằng toán học. Nhưng với cách chứng minh thuật toán đúng bằng toán học thì phức tạp hơn nhiều và đòi hỏi nhiều kiến thức tổng hợp cả về toán học và tin học cộng với khả năng của người thực hiện việc chứng minh thuật toán

3.2. Phân tích tính đơn giản

Đối với các chương trình chỉ dùng 1 vài lần thì yêu cầu giải thuật đơn giản sẽ được ưu tiên vì chúng ta cần 1 giải thuật dễ hiểu, dễ cài đặt, ở đây không đề cao vấn đề thời gian chạy vì chúng ta chỉ chạy 1 vài lần.

Tuy nhiên, khi 1 chương trình sử dụng nhiều lần, yêu cầu tiết kiệm thời gian sẽ được đặc biệt ưu tiên. Tuy nhiên, thời gian thực hiện chương trình lại phụ thuộc vào rất nhiều yếu tố như: cấu hình máy tính, ngôn ngữ sử dụng, trình biên dịch, dữ liệu đầu vào,
… Do đó ta khi so sánh 2 giải thuật đã được implement, chưa chắc chương trình chạy nhanh hơn đã có giải thuật tốt hơn. “Độ phức tạp của thuật toán” sinh ra để giải quyết vấn đề này.

4. Một số giải thuật cơ bản

4.1. Hoán vị hai phần tử

1.Bài toán

INPUT: Nhập giá trị cho hai biến A và B.

OUTPUT: Xuất biến A và B với hai giá trị được hoán đổi. Ví

dụ: Nhập A= 12, B = 50 thì in ra A = 50, B = 12.

2. Trao đổi giá trị của 2 biến A và B thông qua biến trung gian tam :

B0 Bắt đầu

B1 Nhập giá trị cho A và B

B2 Biến tam lấy giá trị của A (Gọi là gán giá trị A cho tam , viết tam := A)

B3 A lấy giá trị của B (Gọi là gán giá trị B cho A , viết A := B)

B4 B lấy giá trị của tam (Gọi là gán giá trị tam cho B , viết B := tam)

B5 Thông báo kết quả

B6 Kết thúc

4.2. Tìm số lớn nhất, nhỏ nhất

Tìm phần tử có giá trị LỚN nhất của dãy số.	**Tìm phần tử có giá trị NHỎ nhất của dãy số.**
**** Ý tưởng:***	**** Ý tưởng:***
+ *giá trị MAX = a_1. Lần lượt với*	+ *giá trị MIN = a_1.*
+ a_i *với Mᴀ N, so sánh số*	+ *với i = 2 đến N, so sánh số a_i với*
nếu a $_i$ *> MAX thì MAX = a_i*	*MIN, nếu a_i > MIN thì MIN = a_i*
□ Xác định bài toán: $_N$	**□ Xác định bài toán:** $_N$
□ ***Input***: N, a_1, a_2, ..., giá trị lớn nhất.	□ ***Input***: N, a_1, a_2, ..., có giá trị nhỏ
□*a* ***Output***:	□*a* ***Output***: nhất.
Xây dựng thuật toán: a_2, ...,	***Xây dựng thuật toán:*** a_2, ...,
Bước 1: Nhập N và dãy a_1, a_N.	*Bước 1: Nhập N và dãy a_1,* a_N.
Bước 2: Max □□□ a_1, i □□ 2;	*Bước 2: Min □□□ □□ 2;*
Bước 3: Nếu i > N thì đưa rồi kết thúc;	*Bước 3: Nếu i > N thì đưa ra giá trị Min rồi kết thúc;*
Bước 4: Nếu ***a_i > Max*** *Max □□□ a_i;*	*Bước 4: Nếu* ***a_i <• Min*** *ì Min □□□ a_i;*
Bước 5: i □□□ quay lại Bước 3;	*Bước 5: i □□□ i quay lại Bước 3;*

4.3. Đệ quy

Thiết kế giải thuật đệ quy

Thực hiện 3 bước sau: _

Tham số hóa bài toán

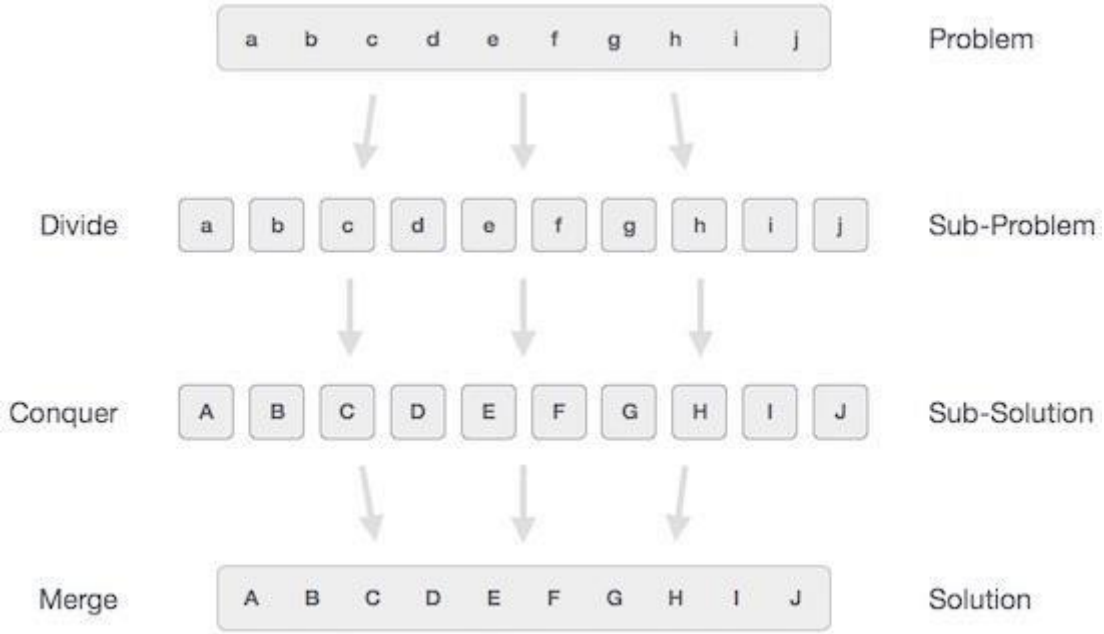
_ Phân tích trường hợp chung: Đưa bài toán về bài toán nhỏ hơn cùng loại, dần dần tiến tới trường hợp suy biến

_ Tìm trường hợp suy biến

4.4. Chia để trị

Giải thuật chia để trị (Divide and Conquer)là gì ?

Phương pháp chia để trị (Divide and Conquer) là một phương pháp quan trọng trong việc thiết kế các giải thuật. Ý tưởng của phương pháp này khá đơn giản và rất dễ hiểu: Khi cần giải quyết một bài toán, ta sẽ tiến hành chia bài toán đó thành các bài toán con nhỏ hơn. Tiếp tục chia cho đến khi các bài toán nhỏ này không thể chia thêm nữa, khi đó ta sẽ giải quyết các bài toán nhỏ nhất này và cuối cùng kết hợp giải pháp của tất cả các bài toán nhỏ để tìm ra giải pháp của bài toán ban đầu.



Nói chung, bạn có thể hiểu giải thuật chia để trị (Divide and Conquer) qua 3 tiến trình sau:

Tiến trình 1: Chia nhỏ (Divide/Break)

- Trong bước này, chúng ta chia bài toán ban đầu thành các bài toán con. Mỗi bài toán con nên là một phần của bài toán ban đầu. Nói chung, bước này sử dụng phương pháp đệ qui để chia nhỏ các bài toán cho đến khi không thể chia thêm nữa. Khi đó, các bài toán con được gọi là "atomic – nguyên tử", nhưng chúng vẫn biểu diễn một phần nào đó của bài toán ban đầu.

Tiến trình 2: Giải bài toán con (Conquer/Solve)

- Trong bước này, các bài toán con được giải.

Tiến trình 3: Kết hợp lời giải (Merge/Combine)

- Sau khi các bài toán con đã được giải, trong bước này chúng ta sẽ kết hợp chúng một cách đệ qui để tìm ra giải pháp cho bài toán ban đầu.

Hạn chế của giải thuật chia để trị (Devide and

Conquer) Giải thuật chia để trị tồn tại hai hạn chế, đó

là:

- Làm thế nào để chia tách bài toán một cách hợp lý thành các bài toán con, bởi vì nếu các bài toán con được giải quyết bằng các thuật toán khác nhau thì sẽ rất phức tạp.
- Việc kết hợp lời giải các bài toán con được thực hiện như thế nào.

BÀI 2: CÁC KIỂU DỮ LIỆU CƠ SỞ

Mã bài: MH09 - 02

Giới thiệu:

Dữ liệu cơ sở là thành phần quan trọng trong việc tạo ra các chương trình, cũng như tạo ra các kiểu dữ liệu mới.

Mục tiêu:

_ Hiểu được khái niệm, phạm vi lưu trữ dữ liệu, các phép xử lý của các kiểu dữ liệu cơ sở như: kiểu số, chuỗi, logic, tập hợp,...;

_ Sử dụng được các kiểu dữ liệu cơ sở trong việc mô tả các đối tượng trong các ngôn ngữ lập trình bậc cao như C, Pascal;

_ Nghiêm túc, tỉ mỉ, sáng tạo trong việc học và vận dụng vào làm bài tập.

Nội dung chính:

1. Các kiểu dữ liệu cơ bản

Kiểu số nguyên là kiểu dữ liệu dùng để lưu các giá trị nguyên hay còn gọi là kiểu đếm được. Kiểu số nguyên trong C được chia thành các kiểu dữ liệu con, mỗi kiểu có một miền giá trị khác nhau

1.1. Kiểu số

1.1.1. Kiểu số nguyên 1 byte (8 bits) *Kiểu số nguyên một byte gồm có 2 kiểu sau:*

1. unsigned char Từ 0 đến 255 (tương đương 256 ký tự trong bảng mã ASCII)
2. char Từ -128 đến 127

Kiểu unsigned char: lưu các số nguyên dương từ 0 đến 255.

=> Để khai báo một biến là kiểu ký tự thì ta khai báo biến kiểu unsigned char. Mỗi số trong miền giá trị của kiểu unsigned char tương ứng với một ký tự trong bảng mã ASCII .

Kiểu char: lưu các số nguyên từ -128 đến 127. Kiểu char sử dụng bit trái nhất để làm bit dấu.

=> Nếu gán giá trị > 127 cho biến kiểu char thì giá trị của biến này có thể là số âm (?).

1.1.2. Kiểu số nguyên 2 bytes (16 bits) *Kiểu số nguyên 2 bytes gồm có 4 kiểu sau:*

1. enum Từ -32,768 đến 32,767
2. unsigned int Từ 0 đến 65,535
3. short int Từ -32,768 đến 32,767
4. int Từ -32,768 đến 32,767

Kiểu enum, short int, int : Lưu các số nguyên từ -32768 đến 32767. Sử dụng bit bên trái nhất để làm bit dấu.

=> Nếu gán giá trị >32767 cho biến có 1 trong 3 kiểu trên thì giá trị của biến này có thể là số âm.

Kiểu unsigned int: Kiểu unsigned int lưu các số nguyên dương từ 0 đến 65535. ***1.1.3. Kiểu số nguyên 4 byte (32 bits)***

Kiểu số nguyên 4 bytes hay còn gọi là số nguyên dài (long) gồm có 2 kiểu sau:

1. unsigned long Từ 0 đến 4,294,967,295
2. long Từ -2,147,483,648 đến 2,147,483,647

Kiểu long : Lưu các số nguyên từ -2147483658 đến 2147483647. Sử dụng bit bên trái nhất để làm bit dấu.

=> Nếu gán giá trị >2147483647 cho biến có kiểu long thì giá trị của biến này có thể là số âm.

Kiểu unsigned long: Kiểu unsigned long lưu các số nguyên dương từ 0 đến 4294967295

Kiểu số thực thường được thực hiện với các phép toán: O =?{+, -, *, /, <, >, <=, >=, =, ?}?

1.1.4. Kiểu số thực: Kiểu số thực dùng để lưu các số thực hay các số có dấu chấm thập phân gồm có 3 kiểu sau:

1. float 4 bytes Từ 3.4 * 10-38 đến 3.4 * 1038
2. double 8 bytes Từ 1.7 * 10-308 đến 1.7 * 10308
3. long double 10 bytes Từ 3.4 *10-4932 đến 1.1 *104932

Mỗi kiểu số thực ở trên đều có miền giá trị và độ chính xác (số số lẻ) khác nhau. Tùy vào nhu cầu sử dụng mà ta có thể khai báo biến thuộc 1 trong 3 kiểu trên. Ngoài ra ta còn có kiểu dữ liệu void, kiểu này mang ý nghĩa là kiểu rỗng không chứa giá trị gì cả.

Kiểu số nguyên thường được thực hiện với các phép toán: O =?{+, -, *, /, DIV, MOD, <, >, <=, >=, =, ?}?

1.2. Kiểu kí tự, chuỗi

+ Kiểu ký tự byte

+ Kiểu ký tự 2 bytes

Kiểu ký tự thường được thực hiện với các phép toán: O =??{+, -, <, >, <=, >=, =, ORD, CHR, ?}?

- Kiểu chuỗi ký tự: Có kích thước tùy thuộc vào từng ngôn ngữ lập trình Kiểu chuỗi ký tự thường được thực hiện với các phép toán: O =??{+,???, <, >, <=, >=, =, Length, Trunc, ?}? - Kiểu luận lý: Thường có kích thước 1 byte

  Kiểu luận lý thường được thực hiện với các phép toán: O =?{NOT, AND, OR, XOR, <, >, <=, >=, =, ?}?

1.3. Kiểu logic

Kiểu **bool** là kiểu dữ liệu chỉ nhận một trong hai giá trị **true** (đúng) hoặc **false** (sai) tương ứng với kết quả của mệnh đề toán học trong C++.

Chúng ta khai báo (và khởi tạo) biến kiểu bool tương tự như cách khai báo biến có các kiểu dữ liệu mà các bạn đã được làm quen. bool b*;*

Trong đó, **bool** là kiểu dữ liệu và **b** là tên biến.

Chúng ta có thể gán trực tiếp giá trị **true** hoặc **false** cho biến kiểu **bool**.

bool b1 = true*;*

bool b2(false);

bool b3 { true }*;*

Giá trị của biến kiểu **bool** có thể bị đảo từ **true** sang **false** hoặc ngược lại nếu sử dụng toán tử **not (!)**.

bool b1 = !true; //**not** true => false bool

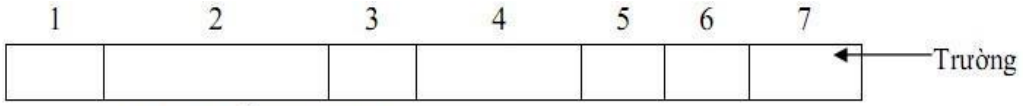
b2(!false); //**not** false => true

2. Kiểu dữ liệu có cấu trúc

2.1 Khái niệm:

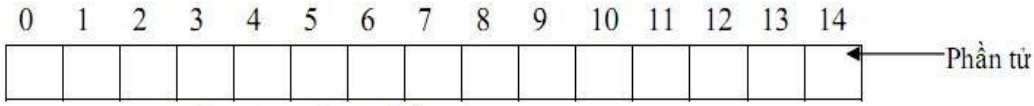
Kiểu cấu trúc (Structure) là kiểu dữ liệu bao gồm nhiều thành phần có kiểu khác nhau, mỗi thành phần được gọi là một trường (field)

Sự khác biệt giữa kiểu cấu trúc và kiểu mảng là: các phần tử của mảng là cùng kiểu còn các phần tử của kiểu cấu trúc có thể có kiểu khác nhau. Hình ảnh của kiểu cấu trúc được minh họa:



Đây là cấu trúc có 7 trường

Còn kiểu mảng có dạng:



Đây là mảng có 15 phần tử

2.2 Định nghĩa kiểu cấu trúc

struct <Tên cấu trúc>

{

<Kiểu> <Trường 1> ; <Kiểu> <Trường 2> ; ……..

<Kiểu> <Trường n> ;

};

Trong đó:

- <Tên cấu trúc>: là một tên được đặt theo quy tắc đặt tên của danh biểu; tên này mang ý nghĩa sẽ là tên kiểu cấu trúc.
- <Kiểu> <Trường i> (i=1..n): mỗi trường trong cấu trúc có dữ liệu thuộc kiểu gì (tên của trường phải là một tên được đặt theo quy tắc đặt tên của danh biểu).

Ví dụ 1: Để quản lý ngày, tháng, năm của một ngày trong năm ta có thể khai báo kiểu cấu trúc gồm 3 thông tin: ngày, tháng, năm.

```
struct NgayThang
{
 unsigned char Ngay;     unsigned char Thang;
unsigned int Nam;
};
type
def
struc
t
{
        unsigned char Ngay;      unsigned char Thang;    unsigned int
Nam;   } NgayThang;
```

Ví dụ 2: Mỗi sinh viên cần được quản lý bởi các thông tin: mã số sinh viên, họ tên, ngày tháng năm sinh, giới tính, địa chỉ thường trú. Lúc này ta có thể khai báo một struct gồm các thông tin trên.

```
struct SinhVien
{
 char MSSV[10];    char HoTen[40];     struct NgayThang NgaySinh;    int
Phai;    char DiaChi[40];
};
type
def
struc
t
{
  char  MSSV[10];         char  HoTen[40];         NgayThang
NgaySinh;      int Phai;     char DiaChi[40];
} SinhVien;
```

2.3 Khai báo biến cấu trúc

Việc khai báo biến cấu trúc cũng tương tự như khai báo biến thuộc kiểu dữ liệu chuẩn. Cú pháp:

- Đối với cấu trúc được định nghĩa theo cách 1: struct <Tên cấu trúc> <Biến 1> [, <Biến 2>…]; - Đối với các cấu trúc được định nghĩa theo cách 2:

<Tên cấu trúc> <Biến 1> [, <Biến 2>…];

Ví dụ: Khai báo biến NgaySinh có kiểu cấu trúc NgayThang; biến SV có kiểu cấu trúc SinhVien. struct NgayThang NgaySinh; struct SinhVien SV;

NgayThang NgaySinh; SinhVien SV;

3. Kiểu tập hợp

3.1. Khái niệm Đối với các kiểu dữ liệu ta đã biết như kiểu số, kiểu mảng, kiểu cấu trúc thì dữ liệu kiểu tập hợp (typedef) là kiểu dữ liệu bao gồm nhiều thành phần có kiểu dữ liệu giống hoặ khác nhau, mỗi thành phần được gọi là một trường (field).

3.2. Các phép xử lý kiểu dữ liệu tập hợp

Sử dụng từ khóa typedef (Type definitions) để định nghĩa kiểu:

```
Typedef  struct
{
   <Kiểu> <Trường 1>  ;
   <Kiểu> <Trường 2>  ;      ……..
   <Kiểu> <Trường n>  ;
} <Tên cấu trúc>;
```

Trong đó:

- typedef (Type definitions): là kiểu do người dùng định nghĩa.
- <Tên cấu trúc>: là một tên được đặt theo quy tắc đặt tên của danh biểu; tên này mang ý nghĩa sẽ là tên kiểu cấu trúc.

<Kiểu> <Trường i> (i=1..n): mỗi trường trong cấu trúc có dữ liệu thuộc kiểu dữ liệu cơ bản.

Ví dụ 1: Để quản lý ngày, tháng, năm của một ngày trong năm ta có thể khai báo kiểu cấu trúc gồm 3 thông tin: ngày, tháng, năm.

```
Typedef  struct
{
        unsigned char Ngay;     unsigned char Thang;   unsigned
        int Nam;
 } NgayThang;
```

Ví dụ 2: Mỗi sinh viên cần được quản lý bởi các thông tin: mã số sinh viên, họ tên, ngày tháng năm sinh, giới tính, địa chỉ thường trú. Lúc này ta có thể khai báo một struct gồm các thông tin trên.

```
typedef struct
{
  char MSSV[10];      char HoTen[40];      NgayThang NgaySinh;
int Phai;     char DiaChi[40];    } SinhVien;
```

3.3. Cài đặt tập hợp

Việc khai báo biến tập hợp cũng tương tự như khai báo biến thuộc kiểu dữ liệu chuẩn.

Cú pháp:

- Đối với cấu trúc được định nghĩa theo cách 1: struct <Tên cấu trúc> <Biến 1> [, <Biến 2>…]; - Đối với các cấu trúc được định nghĩa theo cách 2:

<Tên cấu trúc> <Biến 1> [, <Biến 2>…];

Ví dụ: Khai báo biến NgaySinh có kiểu cấu trúc NgayThang; biến SV có kiểu cấu trúc SinhVien. struct NgayThang NgaySinh; struct SinhVien SV;

NgayThang NgaySinh; SinhVien SV;

BÀI 3: MẢNG, DANH SÁCH VÀ CÁC KIỂU DỮ LIỆU TRỪU TƯỢNG

Mã bài: MH09 - 03

Giới thiệu:

Danh sách là cấu trúc dữ liệu rất thông dụng được cài đặt trên mảng và danh sách liên kết, ngăn xếp và hàng đợi. Đó là các cấu trúc dữ liệu cũng rất gần gũi với các cấu trúc trong thực tiễn.

Mục tiêu:

 Hiểu được khái niệm, cấu trúc lưu trữ của dữ liệu kiểu mảng, kiểu danh sách;

 Hiểu được một số phép toán xử lý trên các phần tử của danh sách liên kết;

 Hiểu cấu trúc, các phép xử lý, khả năng áp dụng của ngăn xếp, hàng đợi;

 Viết được một số giải thuật xử lý các yêu cầu cụ thể trên các kiểu dữ liệu trên; Cài đặt được một số thao tác xử lý danh sách liên kết, ngăn xếp, hàng đợi trên ngôn ngữ C, Pascal;

 Nghiêm túc, tỉ mỉ, sáng tạo trong việc học và vận dụng vào làm bài tập. Chủ động kết hợp các ngôn ngữ lập trình để cài đặt thuật toán.

Nội dung chính:

1. Mảng

1.1. Khái niệm Mỗi biến chỉ có thể biểu diễn một giá trị. Để biểu diễn một dãy số hay một bảng số ta có thể dung nhiều biến nhưng cách này không thuận lợi. trong trường hợp này ta có khái niệm về mảng. khái niệm về mảng trong ngôn ngữ C cũng giống như khái niệm về ma trận trong đại số tuyến tính.

1.2. Cấu trúc lưu trữ của mảng

Mảng có thể hiểu là một tập hợp nhiều phần tử có cùng kiểu giá trị và cùng cùng chung một tên. Mỗi phần tử mảng biểu diễn được một giá trị. Có bao nhiêu kiểu biến thì có bấy nhiêu kiểu mảng. mảng cần được khai báo để định rõ: loại mảng: int, float, double,….

- Tên mảng.
- Số chiều dài và kích thước mỗi chiều.

Khái niệm về kiểu mảng và tên mảng cũng giống như khái niệm về kiểu biến và tên biến. ta sẽ giải thích về số chiều và kích thước mỗi chiều thong qua các ví dụ cụ thể dưới đây. Các khai báo: int a[10],b[4][2]; float x[5],y[3][3]; Chú ý:

Các phần tử của mảng được cấp phát các khoản nhớ lien tiếp nhau trong bộ nhớ. Nói cách khác, các phần tử của mảng liên tiếp nhau.

Trong bộ nhớ, các phần tử của mảng hai chiều được sắp xếp theo hang.

Chỉ số mảng:

Một phần tử cụ thể của mảng được xác định nhờ các chỉ số của nó. Chỉ số của mảng phải có giá trị int không vượt quá kích thước tương ứng. số chỉ số bằng số chiều của mảng.

Giả sử z,b,x,y được khai báo như trên, và giả sử i,j là các biến nguyên trong đó i=2, J=1, khi đó: a[j+i-1] là a[2] b[j+i][2-i] là b[3][0] y[i][j] là y[2][1] Chú ý:

Mảng có bao nhiêu chiều thì ta phải viết bấy nhiêu chỉ số. vì thế nếu ta viết như sau sẽ là sai: y[i](vì y là mảng hai chiều),vv…

Biểu thức dung làm chỉ số có thể thực hiện. khi đó phần nguyên của biểu thức thực sẽ là chỉ số mảng.

Ví dụ:

A[2.5] là a[2]

B[1.9] là a[1]

*Khi chỉ số vượt ra ngoài kích thước mảng, máy sẽ vẫn không báo lỗi, nhưng nó sẽ truy cập đến một vùng nhớ bên ngoài mảng và có thể làm loạn chương trình.

2. Danh sách liên kết

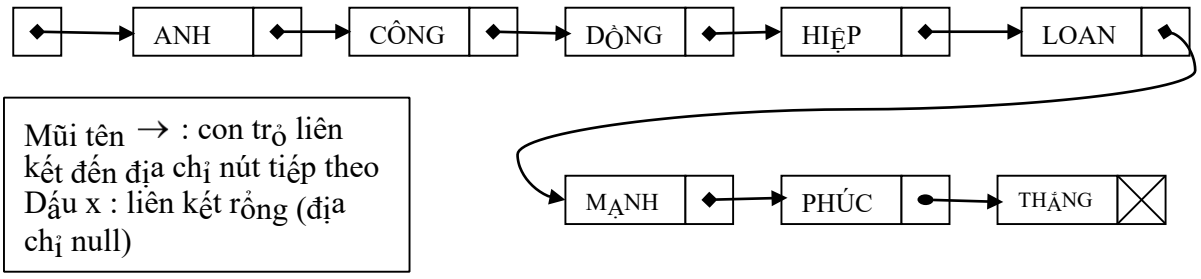
2.1. Danh sách liên kết đơn

Lưu trữ kế tiếp đối với danh sách tuyến tính đã thể hiện rõ nhược điểm trong trường hợp thực hiện thường xuyên các phép bổ sung hoặc loại bỏ phần tử, trường hợp xử lý đồng thời nhiều danh sách v.v...

Việc sử dụng cấu trúc dữ liệu danh sách liên kết để cài đặt kiểu dữ liệu trừu tượng danh sách chính là một giải pháp nhằm khắc phục nhược điểm trên.

Để có thể truy nhập vào mọi nút trong danh sách, ta phải truy nhập từ nút đầu tiên, nghĩa là cần có một con trỏ head trỏ tới nút đầu tiên này.

Có thể minh họa danh sách móc nối này bằng hình ảnh như sau :



Dưới đây là khai báo cấu trúc dữ liệu biểu diễn danh sách liên kết đơn.

Type

Pointer = ^Node ;

```
        Node = Record
                Info : Kieuphantu ;
                Link:    Pointer;
    End ;
```

Ví dụ : Cấu trúc node "SinhVien" như sau:

```
Type   pSinhVien = ^SinhVien
            SinhVien=Record
                MaSV :      String[10];
                 Ten: String[7];
                Diem1,Diem2: Integer;
                DTB: Real;
                Link: PsinhVien;
End;
```

Câu lệnh **call** New(p) sẽ cho ta một nút trống với quy cách ấn định, có địa chỉ là p để sử dụng; còn câu lệnh **call** dispose(p) sẽ trả lại cho "danh sách chỗ trống" nút địa chỉ là p Sau đây ta sẽ xét tới một số giải thuật thực hiện một số phép xử lý trên danh sách móc nối.

1. Khởi tạo danh sách rỗng

Để khởi tạo danh sách rỗng ta chỉ cần lệnh gán:

```
head := NIL;
```

2. Kiểm tra danh sách rỗng

Điều kiện để danh sách liên kết đơn rỗng là head = NIL.

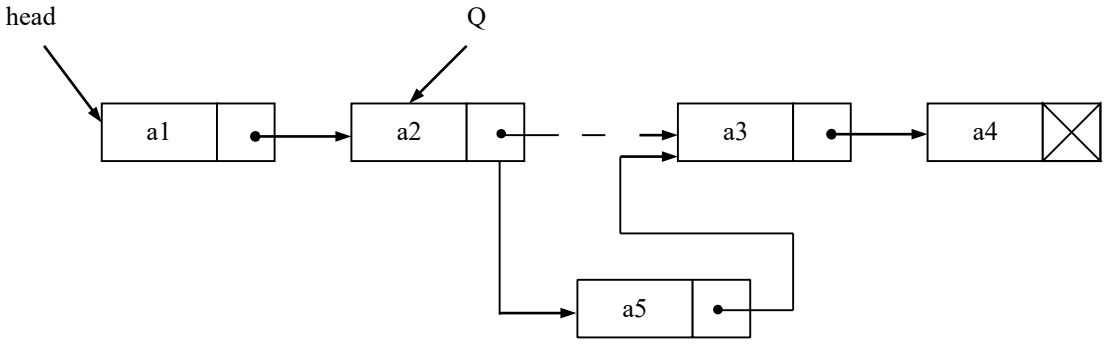
3. Chèn phần tử vào danh sách

Giả sử Q là một con trỏ trỏ vào một nút của danh sách, ta cần bổ sung một nút mới với info là X vào sau nút được trỏ bởi Q. Phép toán này được thực hiện bởi thủ tục sau:

```
Procedure InsertAfter(Var head : Pointer; Q : Pointer; X : Kieuphantu);
    Var  P : Pointer;
    Begin
    {1. Tạo một nút mới}
        NEW(P);
        P^.info := X;
 {Thực hiện bổ sung, nếu danh sách rỗng thì bổ sung nút mới vào thành nút đầu tiên, ngược lại bổ sung nút mới vào sau nút được trỏ bởi Q}
        If head = NIL Then
            begin
                P^.Link := NIL;  head := P;
```

```
            end
        Else   begin
                                    P^.Link := Q^.Link; Q^.Link := P;
            end;
    End;
```

Có thể mô tả thao tác bổ sung trên bằng hình sau.



Hình 3.2

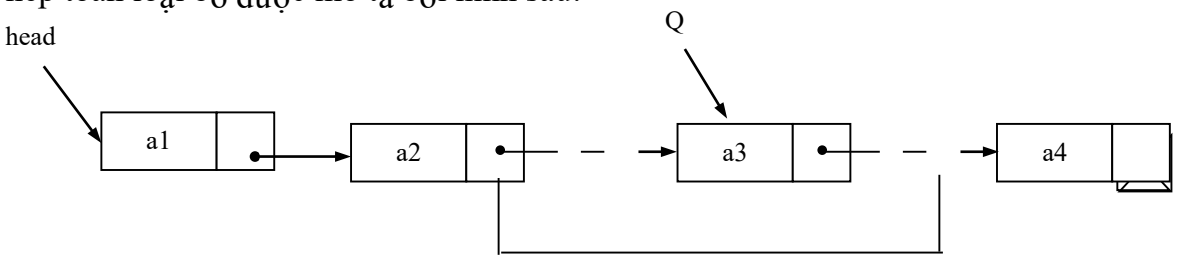
4. Xóa phần tử khỏi danh sách

Cho danh sách liên kết đơn được trỏ bởi head. Q là một con trỏ trỏ vào một nút trong danh sách. Giả sử ta cần loại bỏ nút được trỏ bởi Q. ở đây ta cũng gặp khó khăn là nếu Q không phải là nút đầu tiên thì không xác định được nút đứng trước Q. Trong trường hợp này ta phải tìm đến nút đứng trước Q và cho con trỏ R trỏ vào nút đó, tức là Q = R^.Link. Sau đó ta mới thực hiện loại bỏ nút Q. Ta có thủ tục sau:

```
Procedure Delete(Var head:Pointer; Q:Pointer; Var X:Kieuphantu);
Var R : Pointer;
Begin
{1. Trường hợp danh sách rỗng}
    If head = NIL Then  begin
                            writeln('Danh sach rong');
        Exit;
        end;
    X := Q^.info; {lưu thông tin của nút cần loại bỏ vào biến X}
{2. Trường hợp nút trỏ bởi Q là nút đầu tiên}
    If Q = head Then begin
                head := Q^.Link;
                DISPOSE(Q);
                Exit;
    end;
```

```
{3. Tìm đến nút đứng trước nút trỏ bởi Q}
R := head;
While R^.Link <> Q Do R := R^.Link;
{4. Loại bỏ nút trỏ bởi Q}
R^.Link := Q^.Link;
DISPOSE(Q);
End;
```

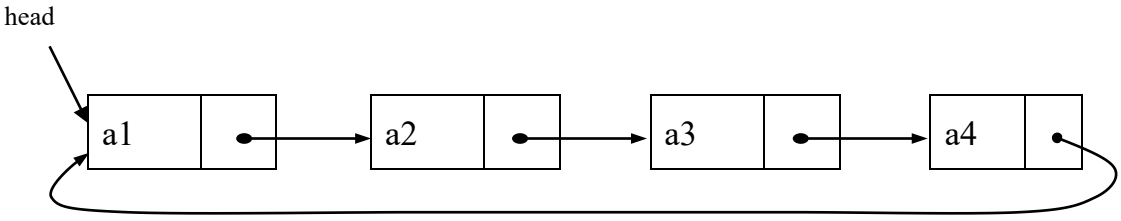
Phép toán loại bỏ được mô tả bởi hình sau:



Hình 3.3

2.2. Danh sách liên kết vòng

Một cải tiến của danh sách liên kết đơn là kiểu danh sách liên kết vòng. Nó khác với danh sách liên kết đơn ở chỗ trường Link của nút cuối cùng trong danh sách không phải bằng NIL, mà nó trỏ đến nút đầu tiên trong danh sách, tạo thành một vòng tròn. Hình ảnh của nó như sau:

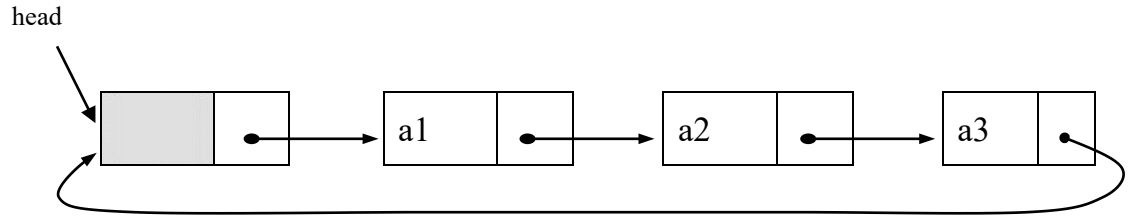


Hình 3.4

Cải tiến này làm cho việc truy nhập vào các nút trong danh sách được linh hoạt hơn. Ta có thể truy nhập vào mọi nút trong danh sách bắt đầu từ nút nào cũng được, không nhất thiết phải từ nút đầu tiên. Điều đó có nghĩa là nút nào cũng có thể coi là nút đầu tiên và con trỏ Head trỏ tới nút nào cũng được. Như vậy, đối với danh sách liên kết vòng chỉ cần cho biết con trỏ trỏ tới nút muốn loại bỏ ta vẫn thực hiện được vì vẫn tìm được đến nút đứng trước đó. Với phép ghép, phép tách cũng có những thuận lợi nhất định.

Tuy nhiên, danh sách nối vòng có một nhược điểm rất rõ là trong khi xử lý, nếu không cẩn thận sẽ dẫn tới một chu trình không kết thúc, bởi vì không biết được vị trí kết thúc danh sách.

Để khắc phục nhược điểm này, người ta đưa thêm vào danh sách một nút đặc biệt gọi là "nút đầu danh sách". Trường info của nút này không chứa dữ liệu của phần tử nào và con trỏ Head bây giờ trỏ tới nút đầu danh sách này. Việc dùng thêm nút đầu danh sách đã khiến cho danh sách về mặt hình thức không bao giờ rỗng. Hình ảnh của nó như sau:

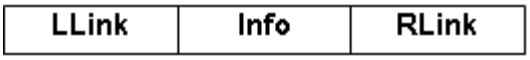


Hình 3.5

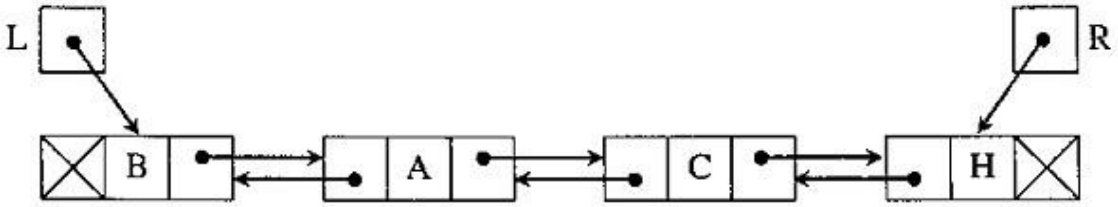
Sau đây là đoạn giải thuật bổ sung một nút vào thành nút đầu tiên trong danh sách có "nút đầu danh sách" trỏ bởi head.

```
New(P);
P^.infor := X;
P^.Link := Head^.Link;
Head^.Link := P;
```

2.3. Danh sách liên kết kép



Khi làm việc với danh sách, có những xử lý trên mỗi nút của danh sách lại liên quan đến cả nút đứng trước và nút đứng sau. Trong những trường hợp như thế, để thuận tiện, người ta đưa vào mỗi nút của danh sách hai con trỏ: LLink trỏ đến nút đứng trước và RLink trỏ đến nút đứng sau nó. Để truy nhập vào danh sách ta dùng hai con trỏ: con trỏ L trỏ vào nút đầu tiên và con trỏ R trỏ vào nút cuối cùng của danh sách. Hình ảnh của danh sách liên kết đôi như sau:

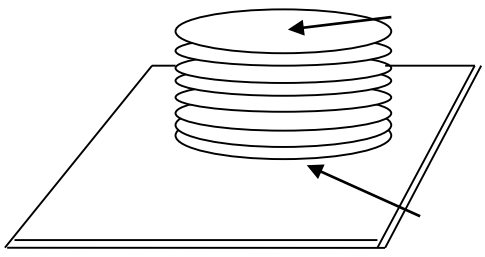


3. Các kiểu dữ liệu trừu tượng

3.1. Ngăn xếp

Ngăn xếp (Stack) là một kiểu danh sách đặc biệt mà phép bổ sung và phép loại bỏ luôn thực hiện ở một đầu ; được gọi là đỉnh.

Có thể hình dung cách tổ chức lưu trữ của stack như một chồng đĩa đặt trên bàn. Đặt thêm một đĩa mới vào thì đặt phía trên đỉnh, lấy một đĩa ra khỏi chồng thì cũng phải lấy ra từ đỉnh. Đĩa đưa vào sau cùng, chính là đĩa đang nằm ở đỉnh, và nó cũng chính là đĩa sẽ lấy ra trước tiên lại đang ở vị trí được gọi là đáy và nó chính là đĩa được lấy ra sau cùng.



Hình 3.7

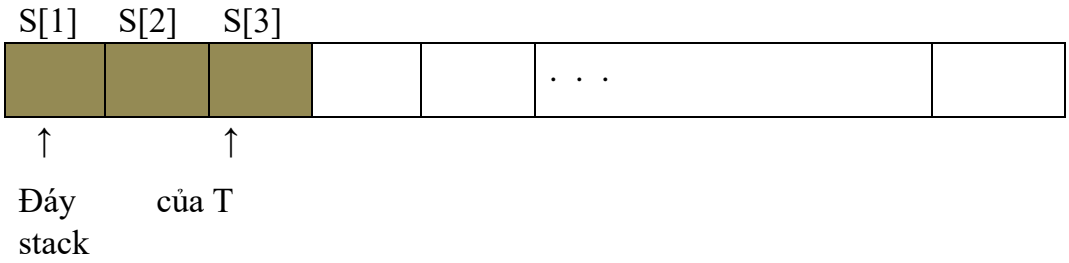
Như vậy stack còn được gọi là danh sách kiểu LIFO (last – in –first –out), tức là stack hoạt động theo cơ chế : "vào – sau – ra – trước"

Biểu diễn stack bằng mảng:

Dùng một mảng để lưu trữ liên tiếp các phần tử của stack. Các phần tử được đưa vào stack bắt đầu từ chỉ số cao nhất của mảng. Chúng ta dùng một biến số nguyên T để lưu trữ chỉ số của phần tử tại đỉnh stack.

Chúng ta quy ước T = 0 nghĩa là stack rỗng. Như vậy T = i thì stack có i phần tử. Rõ ràng $0 \le T \le n$, khi T = n thì stack đã đầy, lúc đó nếu có phép bổ sung một phần tử mới vào stack thì sẽ không thực hiện được, vì "không còn chỗ" ; ta nói là có hiện tượng "tràn" và tất nhiên việc xử lí phải ngừng lại. Còn nếu T =0, nghĩa là stack đã rỗng, mà lại có phép loại bỏ một phần tử ra khỏi stack thì phép xử lí này cũng không thực hiện được; Ta nói có hiện tượng "cạn"

Sau đây là hình ảnh cài đặt của stack với 3 phần tử.



Khi bổ sung một phần tử mới vào thì T tăng lên 1, còn khi loại bỏ một phần tử ra khỏi stack thì T giảm đi 1.

Chúng ta khai báo cấu trúc dữ liệu biểu diễn ngăn xếp như sau:

Const max = N ;

Type Stack =Array[1..max] Of Kieuphantu;

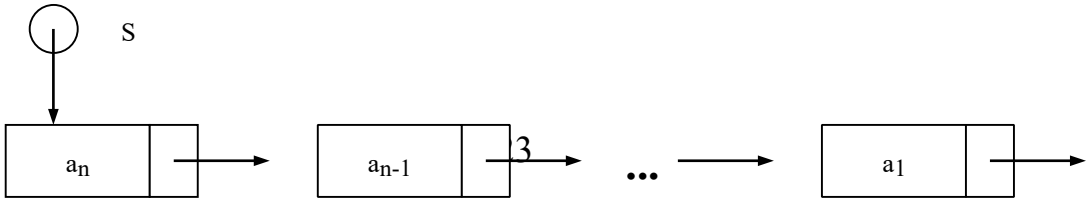
End ;

Var S : Stack ;

Biểu diễn stack bằng danh sách liên kết:

Đối với stack việc truy cập chỉ được thực hiện 1 đầu (đỉnh). Vì vậy, việc cài đặc stack bằng một danh sách nối đơn, có con trỏ head trỏ tới nút đầu tiên, là một cách biểu diễn rất phù hợp. Chúng ta có thể coi head như con trỏ đang trỏ tới đỉnh stack. Bổ sung một nút vào stack chính là bổ sung một nút vào để nó trở thành nút đầu tiên của danh sách, loại bỏ một nút ra khỏi stack chính là loại bỏ nút đầu tiên của danh sách, đang trỏ bởi head. Trong việc bổ sung với ngăn xếp dạng này không cần kiểm tra hiện tượng tràn như với ngăn xếp lưu trữ kế tiếp.

Để cài đặt ngăn xếp bởi danh sách liên kết, ta sử dụng con trỏ S trỏ vào đỉnh của ngăn xếp (hình 3.8). Cấu trúc dữ liệu của ngăn xếp được khai báo như sau:



Hình 3.8

```
Type pointer = ^Node;
 Node = Record
            Info : Kieuphantu;
            Link : pointer;
        End;
Var S : pointer;
```

Trong cách cài đặt này, ngăn xếp rỗng khi S = NIL. Ta giả sử việc cấp phát bộ nhớ động cho các phần tử mới luôn thực hiện. Do đó, ngăn xếp không bao giờ đầy và việc thêm phần tử vào stack luôn thực hiện được.

Sau đây, là các phép xử lý cơ bản trên stack tương ứng với hai cách biểu diễn trên

1. Khởi tạo ngăn xếp rỗng

Cách cài đặt bằng mảng:

```
Procedure Create-Empty(var S : Stack);
    begin
        T := 0;
    end;
```

Cách cài đặt bằng danh sách liên kết đơn:

```
Procedure Create-Empty(var S : pointer);
begin
        T := nil;
    end;
```

2. Kiểm tra ngăn xếp rỗng

Cách cài đặt bằng mảng:

```
Function IsEmpty ( S : Stack) : Boolean;
    begin
            IsEmpty := S[T] = 0;
    end;
```

Cách cài đặt bằng danh sách liên kết đơn:

```
Function IsEmpty ( S : pointer) : Boolean;
begin
            IsEmpty := S[T] = nil;
```

```
end;
```

Hàm IsEmpty nhận giá trị true nếu S rỗng và false nếu S không rỗng.

3. Thêm phần tử vào ngăn xếp

Cách cài đặt bằng mảng:

Để thêm phần tử X vào đỉnh của ngăn xếp S, trước hết phải kiểm tra xem S có đầy không. Nếu S đầy thì việc bổ sung không thực hiện được, ngược lại X được bổ sung vào đỉnh của S.

```
Procedure PUSH ( var S : Stack, X :
Kieuphantu); begin          if  T=n then
begin {kiểm tra stack đầy}
                                writeln(' Stack sẽ
                        tràn'); return;   end;
        else  begin
                                T  :=  T  +  1;  {chuyển  con  trỏ}
                        S[T] := X; {nạp X vào stack}
                        end;
    end;
```

Cách cài đặt bằng danh sách liên kết đơn:

```
    Procedure PUSH ( var S : pointer, X : Kieuphantu);
        Var P : pointer;
        Begin
            {tạo phần tử mới}
            New(P);
            P^.Info := X;
            P^.Link := NIL;
            {thêm phần tử vào đỉnh}
            If S = NIL Then S := P
            Else
            begin
                        P^.Link := S;
                        S    :=    P;
                    end;
        End;
```

4. Xóa phần tử khỏi ngăn xếp

Việc loại bỏ chỉ được thực hiện nếu S không rỗng, giá trị của phần tử bị loại bỏ được gán cho biến X.

Cách cài đặt bằng mảng:

```
procedure POP (var S : Stack; var X : Kieuphantu); begin
    if IsEmpty(S) then begin {kiểm tra stack có rỗng không}
                                write ('Stack đã cạn'); return;
                            end;
        else begin
                X := S[T]; {lưu lại thông tin của phần tử sẽ bị xóa bỏ}
            T := T – 1; {chuyển con trỏ}
        end;
    end;
```

Cách cài đặt bằng danh sách liên kết đơn:

```
    Procedure POP(Var S : pointer; Var X : Kieuphantu;);
    Var P : Tro;
    Begin        if IsEmpty(S) then begin {kiểm tra stack
có rỗng không}
                                write ('Stack đã cạn'); return;
                            end;
                        Else begin {loại bỏ phần tử ra khỏi đỉnh}
                        P := S;
                        X := S^.Info;
                        S := S^.Link;
                        Dispose(P);
        end;
End;
```

3.2. Hàng đợi

Hàng đợi (Queue) là một kiểu danh sách đặc biệt mà phép bổ sung một phần tử vào hàng đợi được thực hiện ở một đầu, gọi là lối sau (rear) và phép loại bỏ một phần tử được thực hiện ở đầu kia, gọi là lối trước (front).

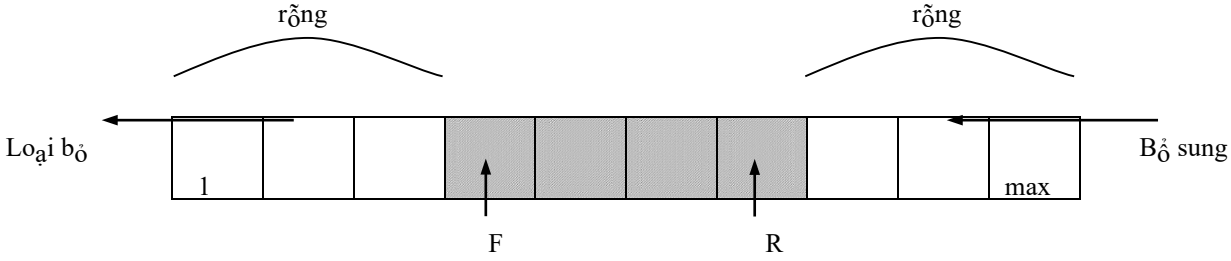
Có thể hình dung cách tổ chức lưu trữ của Queue giống như một hàng đợi: hàng người chờ mua vé tàu, học sinh xếp hàng đi vào lớp v.v...

Bởi vì, queue hoạt động theo cơ chế tự nhiên của nó là vào trước thi ra trước vào sau thì ra sau, cho nên queue còn được gọi là danh sách kiểu FIFO (First – In – First - Out).

Biểu diễn queue bằng mảng:

Tương tự như stasck chúng ta cũng có thể dùng mảng biểu diễn queue với việc sử dụng hai chỉ số F để chỉ vị trí đầu queue (lối trước) và R để chỉ vị trí cuối queue (lối sau). Khi queue rỗng thì F=R=0, nếu thêm phần tử mới vào thì R sẽ tăng lên, nếu xóa bớt phần tử thì F cũng sẽ tăng lên.

Sau đây là hình ảnh minh họa queue:



Hình 3.9: Mảng biểu diễn queue

Chúng ta khai báo cấu trúc dữ liệu biểu diễn queue như sau:

Const max = N ;

Type Queue = Record

F,R : 0..max;

E : Array[1..max] Of Kieuphantu;

End;

Var Q : Queue;

Phương pháp cài đặt hàng đợi với hai chỉ số như trên có nhược điểm lớn. Nếu phép loại bỏ không thường xuyên làm cho queue rỗng, thì các chỉ số F và R sẽ tăng liên tục và sẽ vượt quá cỡ của mảng. Hàng đợi sẽ trở thành đầy, mặc dù các vị trí trống trong mảng có thể vẫn còn nhiều (do việc loại bỏ các phần tử ở đầu hàng).

Để khắc phục nhược điểm trên, chúng ta có thể xem queue như một cấu trúc vòng tròn. Tức là, Q[1] được coi như đứng sau Q[max], phần tử mới được thêm vào hàng đợi tại Q[1]. Xem hình sau:

Biểu diễn queue bằng danh sách liên kết đơn:

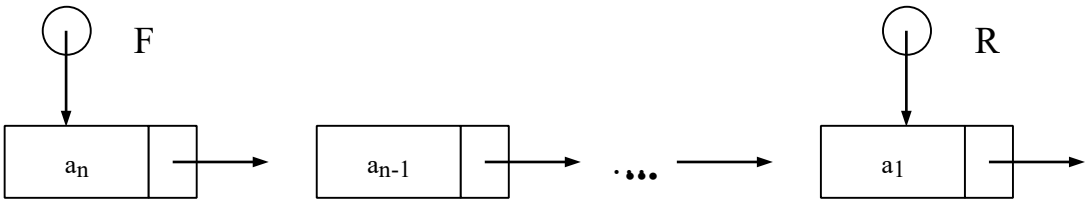
Đối với hàng đợi thì loại bỏ ở một đầu, còn bổ sung thì ở đầu kia. Nếu coi danh sách liên kết đơn như một hàng đợi thì việc loại bỏ một nút tức là loại bỏ nút đầu danh sách, còn việc bổ sung một nút tức là thêm nút mới vào cuối danh sách, nghĩa là phải tìm đến nút cuối cùng. Trong trường hợp này, để lưu trữ danh sách người ta dùng hai con trỏ, một con trỏ trỏ vào nút đầu danh sách và một con trỏ trỏ vào nút cuối danh sách.

Trong trường hợp này, chúng ta sử dụng hai con trỏ, một con trỏ F trỏ đến nút đầu hàng đợi, một con trỏ R trỏ đến nút cuối hàng đợi.

Chúng ta khai báo cấu trúc dữ liệu biểu diễn queue như sau:

```
Type Pointer = ^Node;
 Node = Record
            Info : Kieuphantu;
            Link : pointer;
        End;
 Queue = Record
            F : pointer;
            R : pointer;
        End;
Var  Q : Queue;
```

Sau đây là hình ảnh minh họa queue:



Hình 3.11

Với cách cài đặt này, hàng đợi được xem là không khi nào đầy. Hàng đợi rỗng khi Q^.F = NIL.

Sau đây, là các phép xử lý cơ bản trên queue tương ứng với hai cách biểu diễn trên

1. Khởi tạo hàng đợi rỗng

Cách cài đặt bằng mảng:

```
Procedure Create-Empty(Var Q : Queue);
    Begin
        F := R := 0;
End;
```

Cách cài đặt bằng danh sách liên kết đơn: **Procedure Create-Empty(Var Q : Queue);**

```
Begin
        F := R := 0;
End;
```

2. Kiểm tra hàng đợi rỗng

Cách cài đặt bằng mảng:

```
Function IsEmpty(Q : Queue) : Boolean;
Begin
        IsEmpty := Q[F] = NIL;
End;
```

Cách cài đặt bằng danh sách liên kết đơn:

```
Function IsEmpty(Q : Queue) : Boolean;
Begin
        IsEmpty := Q^.F = NIL;
End;
```

3. Thêm phần tử vào hàng đợi Cách cài đặt bằng mảng:

Để thêm phần tử X vào hàng đợi Q, trước hết phải kiểm tra xem Q có đầy không. Nếu Q đầy thì việc bổ sung không thực hiện được, ngược lại X được bổ sung vào cuối Q.

```
Procedure AddQ(Var Q : Queue; X : Kieuphantu);
Begin
        If  F=1 and R=n or F=R+1 Then {kiểm tra hàng đợi đầy}
write('Queue sẽ tràn')
        If F=0 Then F=R=1 {trước khi thêm vào thì hàng đợi là rỗng}
        Else  if  R=n then R =1
                Else  R := R + 1;
        Q[R] := X;
End;
```

Cách cài đặt bằng danh sách liên kết đơn:

```
Procedure AddQ(Var Q : Queue; X : Kieuphantu);
Var P : pointer;
Begin
        {tạo một phần tử mới}
        New(P);
        P^.Info := X;
        P^.Next := NIL;
```

```
{thêm phần tử mới vào cuối hàng đợi}  If IsEmpty(Q) Then
        begin
                Q^.F := P;
        Q^.R := P;
        end
        Else
    begin
                Q.R^.Link := P;
            Q^.R := P;
    end;
    End;
```

4. Xóa phần tử khỏi hàng đợi Cách cài đặt bằng mảng:

```
Procedure DeleteQ(Var Q : Queue; Var  X : Kieuphantu); Begin
                if  F = nil then write('hang đợi trong')
        else begin
            X := Q[F];
                If  F=R  then F=R=0
                Else  if  F=n then F=1
                Else F=R=1
        End;

End;
```

Cách cài đặt bằng danh sách liên kết đơn:

```
          Procedure Del(Var Q : Queue; Var X : Kieuphantu; Var OK : Boolean);
    Var P : pointer;
    Begin
            If  IsEmpty(Q) Then OK := False
        Else begin
            P := Q^.F;
            X := Q.F^.Info;
            Q^.R := Q.F^.Link;
```

```
                OK := True;                Dispose(P);
                end;
        End;
```

4. Kiểm tra

BÀI 4: CÂY

Mã bài: MH09- 04

Giới thiệu:

Cây là một cấu trúc rất gần gũi và có nhiều ứng dụng trong thực tế. Cây là một cấu trúc phân cấp trên một tập hợp nào đó các đối tượng. Một ví dụ quen thuộc về cây, đó là cây thư mục hoặc mục lục của cuốn sách cũng là một cây. Cây được sử dụng rộng rãi trong rất nhiều vấn đề khác nhau.

Mục tiêu:

- Hiểu các khái niệm, cấu trúc lưu trữ, phân loại, cách duyệt cây;
- Biết nội dung một số bài toán thực tế có thể vận dụng cấu trúc dữ liệu kiểu cây;
- Cài đặt và thực hiện các thao tác trên cây nhị phân;
- Áp dụng cấu trúc dữ liệu dạng cây vào một số bài toán ứng dụng cụ thể như: cây quyết định, mã nén Huffman;
- Nghiêm túc, tỉ mỉ, sáng tạo trong việc học và vận dụng vào làm bài tập.

Nội dung chính:

1. Khái niệm về cây

- Một cây là tập hợp hữu hạn các nút trong đó có một nút đặc biệt gọi là gốc (root). Giữa các nút có một quan hệ phân cấp gọi là "quan hệ cha con".

- Cây được định nghĩa đệ qui như sau:

1. Một nút là một cây và nút này cũng là gỗc của cây.

2. Giả sử $T_1, T_2, \ldots, T_n$ ($n \geq 1$) là các cây có gốc tương ứng $r_1, r_2, \ldots, r_n$. Khi đó cây T với gốc r được hình thành bằng cách cho r trở thành nút cha của các nút $r_1, r_2, \ldots, r_n$

Bậc của một nút: là số con của nút đó

Bậc của một cây: là bậc lớn nhất của các nút có trên cây đó (số cây con tối đa của một nút thuộc cây). Cây có bậc n thì gọi là cây n - phân

Nút gốc: là nút có không có nút cha

Nút lá: là nút có bậc bằng 0

Nút nhánh: là nút có bậc khác 0 và không phải là nút gốc

Mức của một nút

Mức (gốc (T_0)) =1

Gọi $T_1, T_2, \ldots, T_n$ là các cây con của T_0.

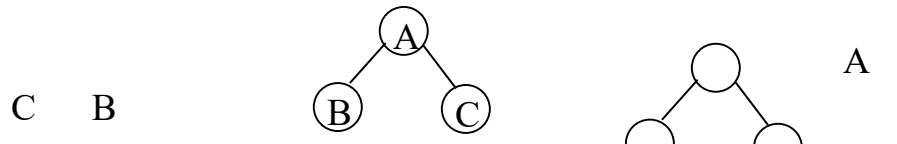
Khi đó Mức (T_1) = Mức (T_2) = ... = Mức (T_n) = Mức (T_0) +1

Chiều cao của cây: là số mức lớn nhất có trên cây đó

Đường đi: Dãy các đỉnh n_1, n_2, ...,n_k được gọi là đường đi nếu n_i là cha của n_{i+1} (1 □ i □ k-1

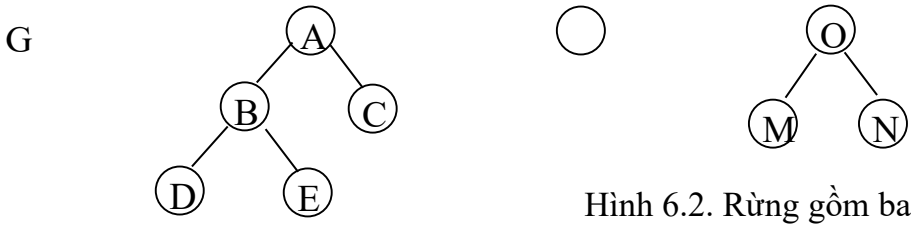
Độ dài của đường đi: là số nút trên đường đi -1

Cây được sắp : Trong một cây, nếu các cây con của mỗi đỉnh được sắp theo một thứ nhất định, thì cây được gọi là cây được sắp (cây có thứ tự). Chẳng hạn, hình minh hoạ hai cây được sắp khác nhau



Hình 6.1. Hai cây được sắp khác nhau

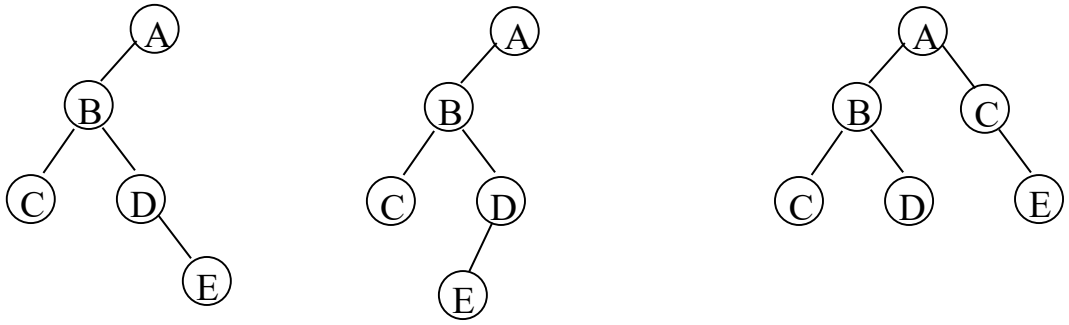
Rừng: là tập hợp hữu hạn các cây phân biệt



Hình 6.2. Rừng gồm ba cây

2. Cây nhị phân

Cây nhị phân là cây mà mỗi nút có tối đa hai cây con. Đối với cây con của một nút người ta cũng phân biệt cây con trái và cây con phải. Như vậy cây nhị phân là cây có thứ tự.



Hình 6.3 . Một số cây nhị phân

Đối với cây nhị phân cần chú ý tới một số tính chất sau

i) Số lượng tối đa các nút có ở mức i trên cây nhị phân là 2^{i-1} (i □ 1) ii) Số lượng nút tối đa trên một cây nhị phân có chiều cao h là 2^h-1(h □ 1)

2.1. Biểu diễn cây nhị phân

Lưu trữ kế tiếp Phương pháp tự nhiên nhất để biểu diễn cây nhị phân là chỉ ra đỉnh con trái và đỉnh con phải của mỗi đỉnh.

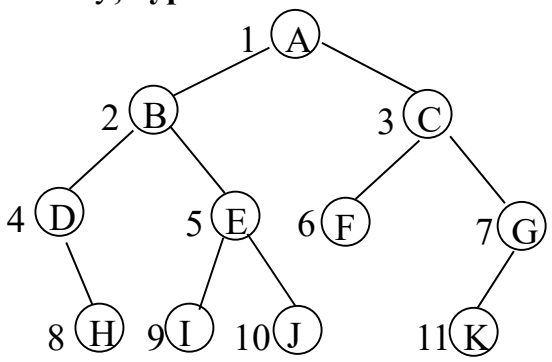
Ta có thể sử dụng một mảng để lưu trữ các đỉnh của cây nhị phân. Mỗi đỉnh của cây được biểu diễn bởi bản ghi gồm ba trường: trường infor: mô tả thông tin gắn với mỗi đỉnh

letf : chỉ đỉnh con trái right:

chỉ đỉnh con phải.

Giả sử các đỉnh của cây được được đánh số từ 1 đến max. Khi đó cấu trúc dữ liệu biểu diễn cây nhị phân được khai báo như sau:

```
const max = ...; {số thứ tự lớn nhất của nút trên cây} type
        item = ...; {kiểu dữ liệu của các nút trên cây}
                Node = record
                infor : item;
                letf :0..max;
right :0..max;
        end;
    Tree = array[1.. max] of Node;
```



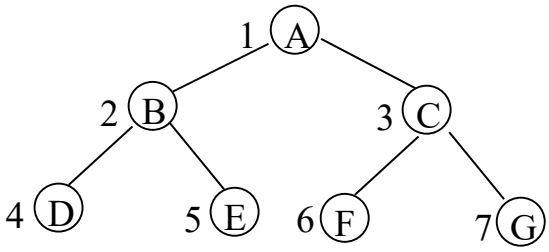
Hình 6.4. Một cây nhị phân

Hình 6.5. Minh hoạ cấu trúc dữ liệu biểu diễn cây nhị phân trong hình 6.4

	infor	left	right
1 2	A	2	3
3 4 5	B	4	5
6	C	6	7
7 8 9	D	0	8
0 1	E	9	10
	F	0	0
	G	11	9
	H	0	0
	I	0	0
	J	0	0
	K	0	0

Hình 6.5. Cấu trúc dữ liệu biểu diễn cây

Nếu có một cây nhị phân hoàn chỉnh đầy đủ, ta có thể dễ dàng đánh số cho các nút trên cây đó theo thứ tự lần lượt từ mức 1 trở lên, hết mức này đến mức khác và từ trái qua phải đối với các nút ở mỗi mức. Ví dụ với hình 5.6 có thể đánh số như sau:



Hình 6.6. Cây nhị phân được đánh số

Ta có nhận xét sau: con của nút thứ i là các nút thứ 2i và 2i + 1 hoặc cha của nút thứ j là □j/2□.

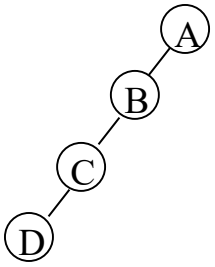
Nếu như vậy thì ta có thể lưu trữ cây này bằng một vectơ V, theo nguyên tắc: nút thứ i của cây được lưu trữ ở V[i]. Đó chính là cách lưu trữ kế tiếp đối với cây nhị phân. Với cách lưu trữ này nếu biết được địa chỉ của nút con sẽ tính được địa chỉ nút cha và ngược lại.

Như vậy với cây đầy đủ nêu trên thì hình ảnh lưu trữ sẽ như sau

A	B	C	D	E	F	G
v[1]	v[2]	v[3]	v[4]	v[5]	v[6]	v[7]

Nhận xét

Nếu cây nhị phân không đầy đủ thì cách lưu trữ này không thích hợp vì sẽ gây ra lãng phí bộ nhớ do có nhiều phần tử bỏ trống (ứng với cây con rỗng). Ta hãy xét cây như hình 5.7. Để lưu trữ cây này ta phải dùng mảng gồm 31 phần tử mà chỉ có 5 phần tử khác rỗng; hình ảnh lưu trữ miền nhớ của cây này như sau:



Hình 6.7.Cây nhị phân đặcbiệt

A	B	□	C	□	□	□	D	□	□	□	□	□	□	□	E	□	...

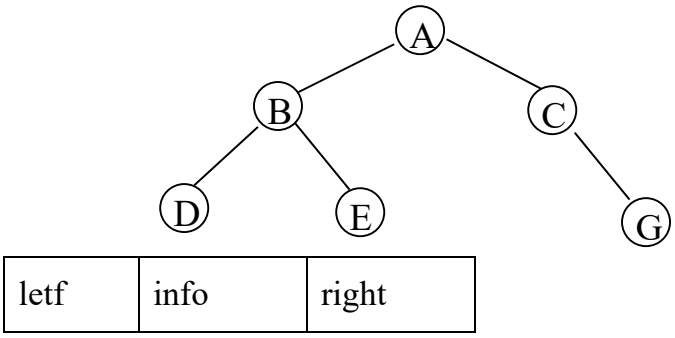
(□: chỉ chỗ trống)

Nếu cây nhị phân luôn biến động nghĩa là có phép bổ sung, loại bỏ các nút thường xuyên tác động thì cách lưu trữ này gặp phải một số nhược điểm như tốn thời gian khi phải thực hiện các thao tác này, độ cao của cây phụ thuộc vào kích thước của mảng...

Lưu trữ móc nối

Cách lưu trữ này khắc phục được các nhược điểm của cách lưu trữ trên đồng thời phản ánh được dạng tự nhiên của cây.

Trong cách lưu trữ này mỗi nút tương ứng với một phần tử nhớ có qui cách như sau:



Trường info ứng với thông tin (dữ liệu) của nút

Trường left ứng với con trỏ, trỏ tới cây con trái của nút đó

Trường right ứng với con trỏ, trỏ

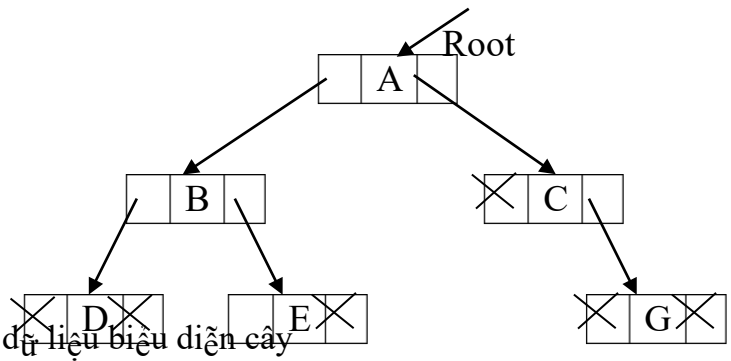
tới cây con phải của nút đó

Hình 6.8

Ta có thể khai báo như sau:

```
        Type                    item   = ...;{kiểu
dữ liệu của các nút trên cây  }
   Tree     =   ^Node;
Node = record    info
: item;      left, right:
Tree;       end;
        var   Root :Tree;
```

Ví dụ: cây nhị phân hình 5.8 có dạng lưu trữ móc nối như ở hình 5.9

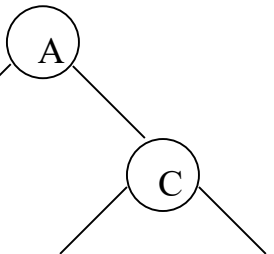


Hình 6.9. Cấu trúc dữ liệu biểu diễn cây

Để truy nhập vào các nút trên cây cần có một con trỏ Root, trỏ tới nút gốc của cây

2.2. Duyệt cây nhị phân

Phép xử lý các nút trên cây - mà ta gọi chung là phép thăm các nút một cách hệ thống, sao cho mỗi nút được thăm đúng một lần, gọi là phép duyệt cây. Chúng ta thường



duyệt cây nhị phân theo một trong ba thứ tự: duyệt trước, duyệt giữa và duyệt sau, các phép này được định nghĩa đệ qui như sau:

2.2.1. Duyệt theo thứ tự trước (gốc – trái – phải)

- Thăm gốc
- Duyệt cây con trái theo thứ trước - Duyệt cây con phải theo thư tự trước

Cài đặt: procedure Truoc(Root : Tree); Begin

```
if Root <> nil then
        Begin write(Root^.info);
              Truoc(Root^.left);
              Truoc(Root^.right); end;
 end;
```

Ví dụ: Chúng ta duyệt trước với cây ở hình 5.12, có kết quả như sau:

A B D C E G H F

2.2.2. Duyệt theo thứ tự giữa (trái – gốc – phải)

- Duyệt cây con trái theo thứ giữa
- Thăm gốc
- Duyệt cây con phải theo thư tự giữa

Cài đặt: procedure Giua(Root : Tree); Begin

```
if Root^.left <> nil then
        Begin
              Preorder(Root^.left); write(Root^.info);
              Preorder(Root^.right); end;
 end;
```

Ví dụ: Chúng ta duyệt trước với cây ở hình 5.12, có kết quả như sau:

D B A G H E C F

2.2.3. Duyệt theo thứ tự sau (trái – phải – gốc)

- Duyệt câycon trái theo thứ sau
- Duyệt cây con phải theo thư tự sau
- Thăm gốc

Cài đặt: procedure

```
Sau(Root : Tree); Begin
    if Root^.right <> nil then
            Begin
```

```
                Preorder(Root^.left);
                Preorder(Root^.right);
                write(Root^.info);
            end;
    end;
```

Ví dụ: Chúng ta duyệt trước với cây ở hình 5.12, có kết quả như sau:

D B H G E F C A

2.3. Cài đặt cây nhị phân

Ta cũng có thể cài đặt cây nhị phân bằng con trỏ bằng cách thiết kế mỗi nút có hai con trỏ, một con trỏ trỏ nút con trái, một con trỏ trỏ nút con phải, trường Data sẽ chứa nhãn của nút.

```
typedef … TData;
typedef struct TNode{TData Data;
TNode* left,right;
};
typedef TNode* TTree;
```

Với cách khai báo như trên ta có thể thiết kế các phép toán cơ bản trên cây nhị phân như sau :

Tạo cây rỗng

Cây rỗng là một cây là không chứa một nút nào cả. Như vậy khi tạo cây rỗng ta chỉ cần cho cây trỏ tới giá trị NULL.

```
void MakeNullTree(TTree *T){
(*T)=NULL;
}
```

Kiểm tra cây rỗng

```
int EmptyTree(TTree T){
return T==NULL;
}
```

Xác định con trái của một nút

```
TTree LeftChild(TTree n){ if (n!=NULL) return n->left; else return NULL; }
```

Xác định con phải của một nút

```
TTree RightChild(TTree
```

```
n){ if (n!=NULL) return n-
>right; else return NULL; }
```

Kiểm tra nút lá:

Nếu nút là nút lá thì nó không có bất kỳ một con nào cả nên khi đó con trái và con phải của nó cùng bằng nil int IsLeaf(TTree n){ if(n!=NULL)

```
return(LeftChild(n)==NULL)&&(RightChild(n)==NULL); else return NULL; }
```

Xác định số nút của cây int

```
nb_nodes(TTree T){
if(EmptyTree(T)) return 0; else return
1+nb_nodes(LeftChild(T))+
nb_nodes(RightChild(T)); }
```

Tạo cây mới từ hai cây có sẵn

```
TTree Create2(Tdata v,TTree l,TTree r){
TTree N;
N=(TNode*)malloc(sizeof(TNode));
N->Data=v;
N->left=l; N-
>right=r; return
N;
}
```

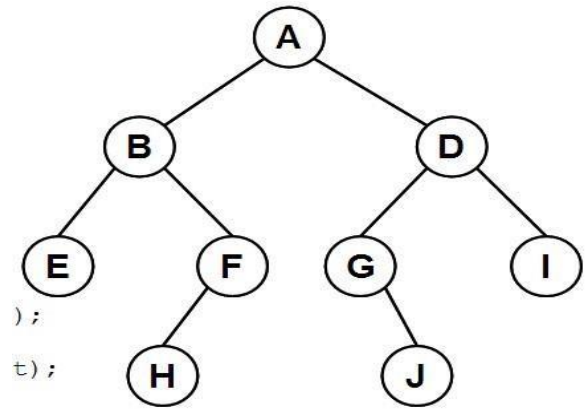
3. Một số bài toán ứng dụng

Duyệt danh sách liên kết là thăm các nút của danh sách từ nút đầu đến nút cuối.

- Duyệt cây là thăm tất cả các nút của cây.
- Có nhiều cách để duyệt cây theo thứ tự duyệt giữa nút cha, nút con bên trái và nút con bên phải.
- Các phép duyệt được thực hiện đệ quy

a. Duyệt theo thứ tự trước NLR (Node – Left – Right)

Thăm nút gốc duyệt cây con bên trái theo NLR duyệt cây con bên phải theo NLR

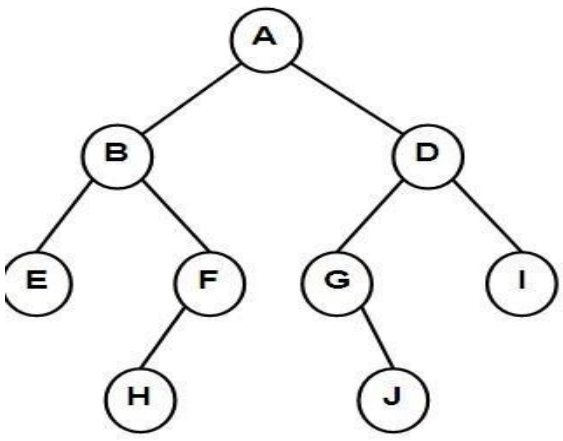


Kết quả duyệt LNR: E B H F A G J D I void DuyetLNR(Tree t)

```
{    if (t!=NULL)
  {
 DuyetLNR(t->pLeft);
 XuLy(t->info);
 DuyetLNR(t->pRight);
  }
```

b. Duyệt theo thứ tự giữa LNR (Left – Node – Right)

Duyệt cây con bên trái theo LNR Thăm nút gốc Duyệt cây con bên phải theo LNR



Ví dụ: xét cây sau

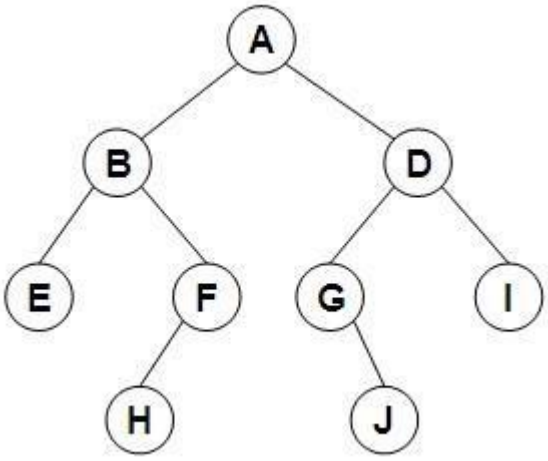
Kết quả duyệt NLR:

A B E F H D G J I void DuyetNLR(Tree t)

```
{   if (t!=NULL)
 {
     XuLy(t->info);
   DuyetNLR(t->pLeft);
   DuyetNLR(t->pRight);
 }
} voi d
ma
in()
{
 DuyetNLR(Root);
}
}
```

c. Duyệt theo thứ tự sau LRN (Left – Right – Node)

Duyệt cây con bên trái theo LRN Thăm nút gốc Duyệt cây con bên phải theo LRN



Kết quả duyệt LRN: E H F B J G I D A void DuyetLRN(Tree t)

```
{    if (t!=NULL)
  {
 DuyetLRN(t->pLeft);
 DuyetLRN(t->pRight);
 XuLy(t->info);
  }
}
```

4. Kiểm tra

BÀI 5: SẮP XẾP

Mã bài: MH09 - 05

Giới thiệu:

Trong chương này chúng ta sẽ được học cài đặt được các thuật toán sắp xếp cơ bản

Mục tiêu:

 Hiểu được tính chất của việc sắp xếp dữ liệu;

 Hiểu được ý tưởng, thuật toán chi tiết của một số phương pháp sắp xếp;

 Áp dụng một số thuật toán sắp xếp vào thực hiện việc sắp xếp các dãy khóa cụ thể;

 Cài đặt được các thuật toán sắp xếp trong ngôn ngữ lập trình bậc cao;

 Nghiêm túc, tỉ mỉ, sáng tạo trong việc học và vận dụng vào làm bài tập.

Nội dung chính:

1. Sắp xếp kiểu chọn, chèn, nổi bọt

1.1. Sắp xếp chọn:

Vấn đề xếp tiền : Có một xấp tiền gồm nhiều tờ có mệnh giá khác nhau đang để lộn xộn, cần

Xếp lại theo thứ tự tiền nhỏ trước, tiền lớn sau.

Phương pháp xếp tiền là: lần lượt ***chọn*** ra các tờ tiền từ nhỏ đến lớn để xếp cho đến khi hết xấp tiền.

Đối với mảng, các bước thực hiện là:

- Trong N phần tử của mảng, chọn phần tử bé nhất, chuyển lên đầu mảng
- Trong N-1 phần tử còn lại, chọn phần tử bé nhất, chuyển vào vị trí thứ 2
- Tiếp tục cho đến khi sắp xếp hết.

1.2. Sắp xếp chèn:

Phương pháp:

- Giống như cách xếp bài khi được chia quân bài.
- Quân bài mới nhận được chèn vào những quân bài đã có trên tay.
- Các quân bài trên tay luôn được sắp xếp.

Thuật toán:

```
void InsertionSort(int a[], int  N)
{  int i, j, temp;
        for(i = 1; i< N; i++)
{
```

```
                temp = a[i];      j
                = i- 1;
                while ((j>=0)&&(a[j]>a[j+1]))
                { a[j+1] = a[j];
                        j--;
                }
if (j!=i-1)                         a[j+1]
= temp;
}
```

1.3. Thuật toán sắp xếp nổi bọt (Bubble Sort):

- **Tư tưởng:**

+ Đi từ cuối mảng về đầu mảng, trong quá trình đi nếu phần tử ở dưới (đứng phía sau) nhỏ hơn phần tử đứng ngay trên (trước) nó thì theo nguyên tắc của bọt khí phần tử nhẹ sẽ bị "trồi" lên phía trên phần tử nặng (hai phần tử này sẽ được đổi chỗ cho nhau). Kết quả là phần tử nhỏ nhất (nhẹ nhất) sẽ được đưa lên (trồi lên) trên bề mặt (đầu mảng) rất nhanh.

+ Sau mỗi lần đi chúng ta đưa được một phần tử trồi lên đúng chỗ. Do vậy, sau N-1 lần đi thì tất cả các phần tử trong mảng M sẽ có thứ tự tăng.

- **Thuật toán:**

B1: First = 1

B2: IF (First = N)

Thực hiện Bkt B3: ELSE

B3.1: Under = N

B3.2: If (Under = First)

Thực hiện B4

B3.3: Else

B3.3.1: if (M[Under] < M[Under - 1])

Swap(M[Under], M[Under - 1])

B3.3.2: Under--

B3.3.3: Lặp lại B3.2 B4: First++ B5: Lặp lại B2 Bkt: Kết thúc

- Cài đặt thuật toán:

Hàm BubbleSort có prototype như sau: void BubbleSort(T

M[], int N);

//Đổi chỗ 2 phần tử cho nhau

Hàm thực hiện việc sắp xếp N phần tử có kiểu dữ liệu T trên mảng M theo thứ tự tăng dựa trên thuật toán sắp xếp nổi bọt. Nội dung của hàm như sau: void BubbleSort(T M[], int N)

```
{   for (int I = 0; I < N-1; I++)   for (int J = N-1; J > I; J--)
        if (M[J] < M[J-1])  Swap(M[J], M[J-1]);
        return;
}
```

Hàm Swap có prototype như sau: void Swap(T????X,
T????Y);

Hàm thực hiện việc hoán vị giá trị của hai phần tử X và Y cho nhau. Nội dung của hàm như sau:

```
void   Swap(T????X, T????Y)
{
        T   Temp = X;
        X = Y;  Y =
        Temp;   return;
}
```

- **Ví dụ minh họa thuật toán:**

Giả sử ta cần sắp xếp mảng M có 10 phần tử sau (N = 10):

M: 15 10 2 20 10 5 25 35 22 30

Ta sẽ thực hiện 9 lần đi (N - 1 = 10 - 1 = 9) để sắp xếp mảng M:

- **Phân tích thuật toán:**

+ Trong mọi trường hợp:

Số phép gán: G = 0

Số phép so sánh: S = (N-1) + (N-2) + ? + 1 = ½N(N-1)

+ Trong trường hợp tốt nhất: khi mảng ban đầu đã có thứ tự tăng

Số phép hoán vị: Hmin = 0

+ Trong trường hợp xấu nhất: khi mảng ban đầu đã có thứ tự giảm

Số phép hoán vị: Hmin = (N-1) + (N-2) + ? + 1 = ½N(N-1)

+ Số phép hoán vị trung bình: Havg = ¼N(N-1)

- **Nhận xét về thuật toán nổi bọt:**

+ Thuật toán sắp xếp nổi bọt khá đơn giản, dễ hiểu và dễ cài đặt.

+ Trong thuật toán sắp xếp nổi bọt, mỗi lần đi từ cuối mảng về đầu mảng thì phần tử nhẹ được trồi lên rất nhanh trong khi đó phần tử nặng lại "chìm" xuống khá chậm chạp do không tận dụng được chiều đi xuống (chiều từ đầu mảng về cuối mảng).

+ Thuật toán nổi bọt không phát hiện ra được các đoạn phần tử nằm hai đầu của mảng đã nằm đúng vị trí để có thể giảm bớt quãng đường đi trong mỗi lần

2. Sắp xếp kiểu phân đoạn

Phương pháp:

Dùng giải pháp đệ quy (chia để trị)

_ *Bước 1*: Phân hoạch mảng A ban đầu thành 2 mảng con B và C sao cho b_i

- c_j b_i □ B, c_j □ C. □

_ *Bước 2*: Sắp xếp mảng con B bằng đệ quy

_ *Bước 3*: Sắp xếp mảng con C bằng đệ quy

Điều kiện dừng: khi mảng con cần sắp chỉ có 1 phần tử □ xem như được sắp.

_ Vì B, C được sắp và b_i c_j nên □ mảng A là được sắp

3. Sắp xếp kiểu hòa nhập

Phương pháp:

Cũng sử dụng giải pháp chia để trị

- *Bước 1*: Chia mảng A ban đầu thành 2 mảng con B và C.
  - *Bước 2, 3*: Sắp xếp mảng con B và C bằng đệ quy (Điều kiện dừng: khi mảng con cần sắp chỉ có 1 phần tử)
- *Bước 4*: Trộn (merge) 2 mảng con đã sắp B, C thành mảng A được sắp.

Thuật toán:

```
int Partition(int a[], int p, int r)
{       int t;     // phân
hoạch           return t;
}

void QuickSort(int a[], int p, int r)
{       int t = Partition(a, p, r);      if
(p< t-1) QuickSort(a, p, t-1);      if
(t+1< r) QuickSort(a, t+1, r);
```

}

4. Kiểm tra

BÀI 6: TÌM KIẾM

Mã bài: MH09 - 06

Giới thiệu:

Trong hầu hết các hệ lưu trữ, quản lý dữ liệu, thao tác tìm kiếm thường được thực hiện nhiều nhất để khai thác thông tin. Các thuật toán sắp xếp và tìm kiếm cùng với các kỹ thuật được sử dụng trong đó được coi là các kỹ thuật cơ sở cho lập trình máy tính.

Mục tiêu:

_ Hiểu được ý tưởng, thuật toán chi tiết của một số phương pháp tìm kiếm;

_ Áp dụng một số thuật toán tìm kiếm vào các dãy khóa cụ thể;

_ Cài đặt được các thuật toán tìm kiếm trong ngôn ngữ lập trình bậc cao;

_ Nghiêm túc, tỉ mỉ, sáng tạo trong việc học và vận dụng vào làm bài tập.

Nội dung chính:

1. Tìm kiếm tuần tự

Thuật toán tìm tuyến tính còn được gọi là Thuật toán tìm kiếm tuần tự (Sequential Search).

a. Tư tưởng:

Lần lượt so sánh các phần tử của mảng M với giá trị X bắt đầu từ phần tử đầu tiên cho đến khi tìm đến được phần tử có giá trị X hoặc đã duyệt qua hết tất cả các phần tử của mảng M thì kết thúc. **b. Thuật toán:**

B1: k = 1

B2: IF M[k]?? X AND k?? N B2.1:

k++

B2.2: Lặp lại B2 B3: IF k?? N

Tìm thấy tại vị trí k B4: ELSE

//Duyệt từ đầu mảng

//Nếu chưa tìm thấy và cũng chưa duyệt hết mảng

Không tìm thấy phần tử có giá trị X

B5: Kết thúc

c. Cài đặt thuật toán:

Hàm LinearSearch có prototype: int LinearSearch (T M[], int N, T X);

Hàm thực hiện việc tìm kiếm phần tử có giá trị X trên mảng M có N phần tử. Nếu tìm thấy, hàm trả về một số nguyên có giá trị từ 0 đến N-1 là vị trí tương ứng của phần tử

tìm thấy. Trong trường hợp ngược lại, hàm trả về giá trị -1 (không tìm thấy). Nội dung của hàm như sau:

```
int   LinearSearch (T   M[], int   N, T   X)
{    int k = 0;   while (M[k] !=
        X??? k < N)   k++;
        if (k < N)  return (k);   return
        (-1);
}
```

d. Phân tích thuật toán:

- Trường hợp tốt nhất khi phần tử đầu tiên của mảng có giá trị bằng X:

Số phép gán: Gmin = 1

Số phép so sánh: Smin = 2 + 1 = 3

- Trường hợp xấu nhất khi không tìm thấy phần tử nào có giá trị bằng X:

Số phép gán: Gmax = 1 Số phép so sánh: Smax = 2N+1 - Trung bình:

Số phép gán: Gavg = 1

Số phép so sánh: Savg = (3 + 2N + 1) : 2 = N + 2 **e.**

Cải tiến thuật toán:

Trong thuật toán trên, ở mỗi bước lặp chúng ta cần phải thực hiện 2 phép so sánh để kiểm tra sự tìm thấy và kiểm soát sự hết mảng trong quá trình duyệt mảng. Chúng ta có thể giảm bớt 1 phép so sánh nếu chúng ta thêm vào cuối mảng một phần tử cầm canh (sentinel/stand by) có giá trị bằng X để nhận diện ra sự hết mảng khi duyệt mảng, khi đó thuật toán này được cải tiến lại như sau:

B1: k = 1

B2: M[N+1] = X

B3: IF M[k]?? X B3.1: k++

B3.2: Lặp lại B3 B4: IF k < N

Tìm thấy tại vị trí k B5: ELSE //Phần tử cầm canh

//k = N song đó chỉ là phần tử cầm canh

Không tìm thấy phần tử có giá trị X

B6: Kết thúc

Hàm LinearSearch được viết lại thành hàm LinearSearch1 như sau:

```
int   LinearSearch1 (T   M[], int   N, T   X)
```

```
{   int k = 0;   M[N] = X;
        while (M[k] != X)
        k++;
        if  (k  <  N)    return  (k);
        return (-1);
}
```

f. Phân tích thuật toán cải tiến:

- Trường hợp tốt nhất khi phần tử đầu tiên của mảng có giá trị bằng X:

Số phép gán: Gmin = 2

Số phép so sánh: Smin = 1 + 1 = 2

- Trường hợp xấu nhất khi không tìm thấy phần tử nào có giá trị bằng X:

Số phép gán: Gmax = 2

Số phép so sánh: Smax = (N+1) + 1 = N + 2 - Trung bình:

Số phép gán: Gavg = 2

Số phép so sánh: Savg = (2 + N + 2) : 2 = N/2 + 2

- Như vậy, nếu thời gian thực hiện phép gán không đáng kể thì thuật toán cải tiến sẽ chạy nhanh hơn thuật toán nguyên thủy.

2. Tìm kiếm nhị phân

Thuật toán tìm tuyến tính tỏ ra đơn giản và thuận tiện trong trường hợp số phần tử của dãy không lớn lắm. Tuy nhiên, khi số phần tử của dãy khá lớn, chẳng hạn chúng ta tìm kiếm tên một khách hàng trong một danh bạ điện thoại của một thành phố lớn theo thuật toán tìm tuần tự thì quả thực mất rất nhiều thời gian. Trong thực tế, thông thường các phần tử của dãy đã có một thứ tự, do vậy thuật toán tìm nhị phân sau đây sẽ rút ngắn đáng kể thời gian tìm kiếm trên dãy đã có thứ tự. Trong thuật toán này chúng ta giả sử các phần tử trong dãy đã có thứ tự tăng (không giảm dần), tức là các phần tử đứng trước luôn có giá trị nhỏ hơn hoặc bằng (không lớn hơn) phần tử đứng sau nó.

Khi đó, nếu X nhỏ hơn giá trị phần tử đứng ở giữa dãy (M[Mid]) thì X chỉ có thể tìm thấy ở nửa đầu của dãy và ngược lại, nếu X lớn hơn phần tử M[Mid] thì X chỉ có thể tìm thấy ở nửa sau của dãy. **a. Tư tưởng:**

Phạm vi tìm kiếm ban đầu của chúng ta là từ phần tử đầu tiên của dãy (First = 1) cho đến phần tử cuối cùng của dãy (Last = N). So sánh giá trị X với giá trị phần tử đứng ở giữa của dãy M là M[Mid].

Nếu X = M[Mid]: Tìm thấy

Nếu X < M[Mid]: Rút ngắn phạm vi tìm kiếm về nửa đầu của dãy M (Last = Mid-

1) Nếu X > M[Mid]: Rút ngắn phạm vi tìm kiếm về nửa sau của dãy M (First = Mid+1)

Lặp lại quá trình này cho đến khi tìm thấy phần tử có giá trị X hoặc phạm vi tìm kiếm của chúng ta không còn nữa (First > Last).

b. Thuật toán đệ quy (Recursion Algorithm):

B1: First = 1

B2: Last = N

B3: IF (First > Last)

B3.1: Không tìm thấy

B3.2: Thực hiện Bkt

B4: Mid = (First + Last)/ 2 B5: IF (X = M[Mid]) //Hết phạm vi tìm kiếm

B5.1: Tìm thấy tại vị trí Mid

B5.2: Thực hiện Bkt

B6: IF (X < M[Mid])

Tìm đệ quy từ First đến Last = Mid - 1 B7: IF (X > M[Mid])

Tìm đệ quy từ First = Mid + 1 đến Last

Bkt: Kết thúc

c. Cài đặt thuật toán đệ quy:

Hàm BinarySearch có prototype: int BinarySearch (T M[], int N, T X); Hàm thực hiện việc tìm kiếm phần tử có giá trị X trong mảng M có N phần tử đã có thứ tự tăng. Nếu tìm thấy, hàm trả về một số nguyên có giá trị từ 0 đến N-1 là vị trí tương ứng của phần tử tìm thấy. Trong trường hợp ngược lại, hàm trả về giá trị -1 (không tìm thấy). Hàm BinarySearch sử dụng hàm đệ quy RecBinarySearch có prototype: int RecBinarySearch(T M[], int First, int Last, T X);

Hàm RecBinarySearch thực hiện việc tìm kiếm phần tử có giá trị X trên mảng M trong phạm vi từ phần tử thứ First đến phần tử thứ Last. Nếu tìm thấy, hàm trả về một số nguyên có giá trị từ First đến Last là vị trí tương ứng của phần tử tìm thấy. Trong trường hợp ngược lại, hàm trả về giá trị -1 (không tìm thấy).

3. Cây tìm kiếm nhị phân

3.1. Định nghĩa Cây tìm kiếm ứng với n khóa {\displaystyle k_{1},k_{2},...k_{n}} {\displaystyle k_{1},k_{2},...k_{n}} là cây nhị phân mà mỗi nút đều được gán một khóa sao cho với mỗi mỗi nút k:

Mọi khóa trên cây con trái đều nhỏ hơn khóa trên nút k

Mọi khóa trên cây con phải đều lớn hơn khóa trên nút k

Cây tìm kiếm nhị phân là một cấu trúc dữ liệu cơ bản được sử dụng để xây dựng các cấu trúc dữ liệu trừu tượng hơn như các tập hợp, đa tập hợp, các dãy kết hợp.

Nếu một BST có chứa các giá trị giống nhau thì nó biểu diễn một đa tập hợp. Cây loại này sử dụng các bất đẳng thức không nghiêm ngặt. Mọi nút trong cây con trái có khóa nhỏ hơn khóa của nút cha, mọi nút trên cây con phải có nút lớn hơn hoặc bằng khóa của nút cha.

Nếu một BST không chứa các giá trị giống nhau thì nó biểu diễn một tập hợp đơn trị như trong lý thuyết tập hợp. Cây loại này sử dụng các bất đẳng thức nghiêm ngặt.

Mọi nút trong cây con trái có khóa nhỏ hơn khóa của nút cha, mọi nút trên cây con phải có nút lớn hơn khóa của nút cha.

Việc chọn đưa các giá trị bằng nhau vào cây con phải (hay trái) là tùy theo mỗi người. Một số người cũng đưa các giá trị bằng nhau vào cả hai phía, nhưng khi đó việc tìm kiếm trở nên phức tạp hơn.

3.2. Cài đặt cây tìm kiếm nhị phân

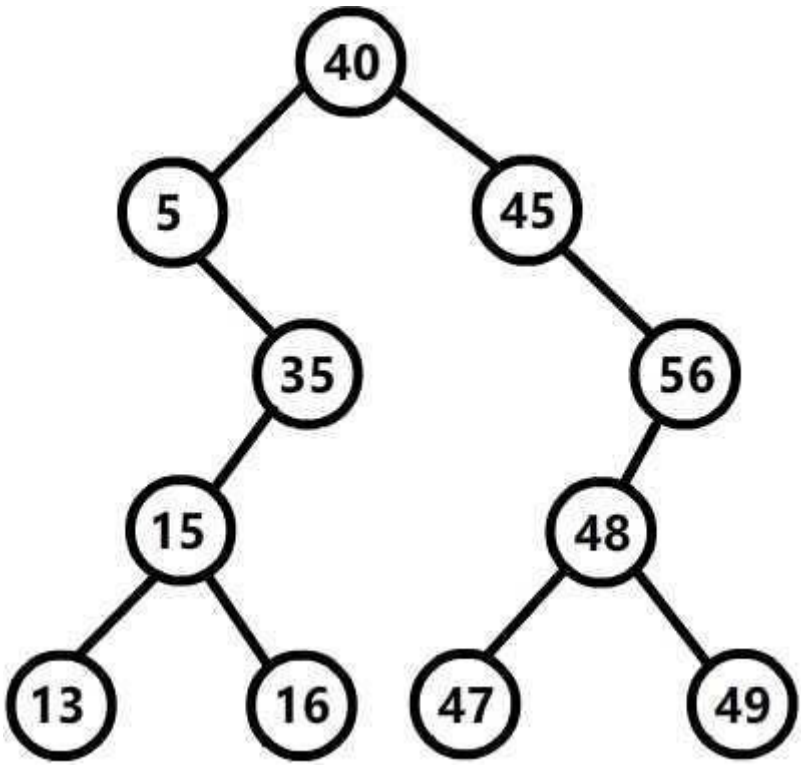
Khởi tạo cây (init)

- Khởi tạo node: cấp phát bộ nhớ cho 1 node, truyền data cần lưu trữ, gán pLeft = pRight = NULL
- Khởi tạo cây : khởi tạo cây và gán root = NULL.

```
1.  struct NODE
2.  {
3.  int data;
4.  NODE* pLeft;
5.  NODE* pRight;
6.  };
7.  NODE* CreateNode(int x)
8.  {
9.  NODE* p = new NODE();
10. p->data = x;
11. p->pLeft = p->pRight = NULL;
12. return p;
13. }
```

Chèn (insertion)

Chèn node chứa dữ liệu vào cây có gốc là root và trả về địa chỉ node mới chèn, việc chèn dữ liệu phải dựa trên đặc điểm của cây nhị phân tìm kiếm. VD: Input: { 40, 5, 35, 45, 15, 56, 48, 13, 16, 49, 47 };



- Tìm vị trí node cần chèn :

```
1.  NODE* FindInsert(NODE* root, int x)
2.  {
3.  if (root == NULL)
4.  {
5.  return NULL;
6.  }
7.  NODE* p = root;
8.  NODE* f = p;
9.  while (p != NULL)
10. {
11. f = p;
12. if (p->data > x)
13. {
14. p = p->pLeft;
15. }
16. else
17. {
18. p = p->pRight;
```

```
19.         }
20.         }
21.         return f;
22.         }
```


- Chèn node :

```
1.          void InsertNode(NODE* &root, int x)
2.          {
3.          NODE* n = CreateNode(x);
4.          if (root == NULL)
5.          {
6.          root = n;
7.          return;
8.          }
9.          else
10.         {
11.         NODE* f = FindInsert(root, x);
12.         if (f != NULL)
13.         {
14.         if (f->data > x)
15.         {
16.         f->pLeft = n;
17.         }
18.         else
19.         {
20.         f->pRight = n;
21.         }
22.         }
23.         }
24.         }
```

Tạo cây nhị phân tìm kiếm

```
1.  void CreateTree(NODE* &root, int a[], int n)
2.  {
3.  for (int i = 0; i < n; i++)
4.  {
5.  InsertNode(root, a[i]);
6.  }
7.  }
```

Tìm kiếm (searching)

Tìm node "x" trên cây root, trả về địa chỉ nếu tìm thấy node x. □ Sử dụng đệ quy :

```
1.  NODE* SearchNode_Re(NODE* root, int x)
2.  {
3.  if (root == NULL) 4.  return NULL;
5.
6.  if (root->data == x)
7.  {
8.  return root;
9.  }
10.  if (root->data > x)
11.  {
12.  SearchNode_Re(root->pLeft, x);
13.  }
14.  else
15.  {
16.  SearchNode_Re(root->pRight, x);
17.  }
18.  }
```

- Sử dụng vòng lặp :

```
1.  NODE* SearchNode(NODE* root, int x)
2.  {
```

```
3.      if (root == NULL)
4.      return NULL;
```



```
5.
6.              NODE* p = root;
7.              while (p != NULL)
8.              {
9.              if (p->data == x)
10.             {
11.             return p;
12.             }
13.             else if (p->data > x)
14.             {
15.             p = p->pLeft;
16.             }
17.             else
18.             {
19.             p = p->pRight;
20.             }
21.             }
22.             }
```

4. Kiểm tra

TÀI LIỆU THAM KHẢO

- Đỗ Xuân Lôi, *Cấu trúc dữ liệu và giải thuật*, NXB Thống kê, 1999;
- Hoàng Nghĩa Tý, *Cấu trúc dữ liệu và thuật toán*, NXB Xây dựng, 2000.